Proceedings

of the ECAI 2000 Workshop on

“New Results in Planning, Scheduling, and Design”

PuK 2000

Edited by
Jürgen Sauer, Jana Köhler

Berlin 21.-22.8.2000

Workshop Organization

Dr. Jürgen Sauer
Universität Oldenburg, FB Informatik
Escherweg 2
D-26121 Oldenburg, Germany
Tel.: ++49-441-9722-220
Fax: ++49-441-9722-202
e-mail: sauer@informatik.uni-oldenburg.de
www: http://www-is.informatik.uni-oldenburg.de/~sauer/

Dr. Jana Köhler
Schindler Lifts Ltd
R & D Technology Management
CH-6031 Ebikon, Switzerland
Tel.: ++41-41-445-4840
Fax: ++41-41-445-5297
e-mail: jana_koehler@ch.schindler.com

Program Committee

Preface

The areas of planning, scheduling, and design are sharing methodologies and often use the same or similar AI based techniques. However, the communities do not overlap and joint meetings are rather rare. The workshop therefore aims to provide a platform for the exchange of ideas, concepts, and problems between researchers and practitioners from the areas of planning, scheduling, design and configuration.

The workshop will be held from **21. - 22. August 2000 in Berlin** as one of the workshops at the European Conference on Artificial Intelligence (ECAI 2000). It is also the 14th in a series of workshops of the special interest group on planning, scheduling and design (PuK) of the German Gesellschaft für Informatik. The workshop will also be organized as a PLANET related meeting and receive special support by PLANET (the european network of excellence in planning (link: http://planet.dfki.de/)).

Within the workshop researchers and practitioners present and discuss new approaches, systems and problem areas in planning, scheduling, design and configuration. Traditionally, the focus is on AI related topics including knowledge representation and problem solving as well as system design.

Topics of interest include but are not limited to:

- Applications and architectures:
  empirical studies of existing systems; new and prototypical systems; modeling of planning/ scheduling/ design systems; user interfaces.
- Knowledge representation and problem solving techniques:
  domain-specific techniques; heuristic techniques; distributed problem solving; constraint-based techniques; iterative improvement; integrating reaction and user-interaction.
- Learning:
  learning in the context of planning, scheduling and design.

From the papers submitted the international program committee has selected 18 for presentation in the workshop. The papers provide a good mix of planning and scheduling as well as design related topics. We hope that the ideas and approaches presented in the papers and presentations will lead to a fruitful discussion and will inspire research and development in all of the participating research directions.

Berlin, August 2000

Jürgen Sauer, Jana Köhler

Contents

A hybrid hierarchical/operator-based planning approach for the design of control programs

L. Castillo and **J. Fdez-Olivares** and **A. González** [1]

Abstract.

The design of a control program is a complex process whose result must satisfy very restrictive constraints imposed by new manufacturing systems needs as flexibility, quick response, correctness and low-cost building process. Current AI Planning approaches for the synthesis of control programs are proving to be very useful to satisfy these needs. But they have to be extended in order to build efficient and realistic systems which obtain truly real world solutions. This work presents an approach in this direction which mixes hierarchical and POCL techniques in order to build an architecture closer to the way that control engineers reason in order to design a control program. The utility of this approach is shown along this paper.

1 INTRODUCTION

The design of a correct and complete industrial control program is a process which involves different sources of knowledge and whose final result is a sequence of control actions [6]. Concurrency, conditional branches, soundness, security and flexibility are some of the features that these sequences are expected to have.

The design process of a control program with these features is very complex, even for human programmers. Traditionally control engineers use different methodologies, standards, formal tools and computer utilities to carry out this task. The ISA-SP88 [13] standard (Figure 1) is one of such methodologies used to hierarchically design control programs for manufacturing systems. This standard allows for a hierarchical specification of physical, process and control models of a manufacturing system.

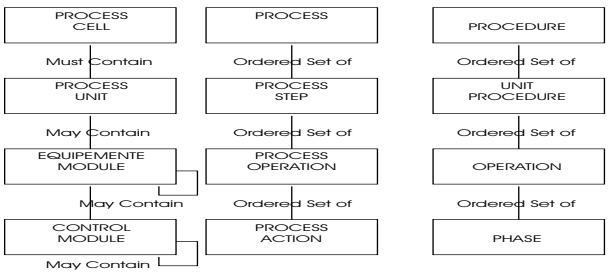


Figure 1. Physical, Process and Control Model of SP88.

[1] Departamento de Ciencias de la Computación e Inteligencia Artificial, E.T.S. Ingeniería Informática. Universidad de Granada, 18071 Granada, SPAIN, email: L.Castillo,Faro,A.Gonzalez@decsai.ugr.es

Starting from a hierarchical physical model and from a process specification (recipe) at the higher abstraction level, a control engineer obtains a modular control program at different levels of granularity. There are formal tools as GRAFCET [12] for the representation and specification of such programs.

These methodologies used by control engineers to develop control programs are useful and necessary, buy they are not sufficient if we take into account requirements as flexibility and quick response in new generation manufacturing systems [7]. The application of AI Planning techniques for the synthesis of control programs or operating procedures in manufacturing systems is a promising technology that meets these requirements. Although at present it is in an initial stage [5], interesting approaches have been carried out ([1, 6, 14, 20]). These techniques are proving to be very useful, allowing for an error-free, fast and low-cost building process of control programs.

Not all of these approaches are based on hierarchical planning techniques [8, 10, 18], which are useful to represent the hierarchical structure of devices and their operation in manufacturing systems. In addition, hierarchical techniques are closer to the way in that control engineers

- represent a control program (modular and hierarchically), and
- reason in order to find the sequence of instructions of the control program.

In this sense, this work presents a planning approach which employs hybrid POCL and hierarchical planning techniques in order to

- Represent an industrial plant as a device hierarchy at different levels of granularity, which accepts SP88 descriptions, providing a friendly input level for control engineers, and
- autonomously develop control programs for manufacturing systems following a hybrid planning process (POCL+hierarchical), which results in a hierarchy of control sequences (plans) at different levels of detail, closer to the way that humans develop modular industrial control programs and, thus, providing a more understandable output.

In the next section we will show some related work and, afterwards, we will describe our approach.

2 RELATED WORK

Apart from the partial-order planning approaches mentioned in previous section, one of the more recent hierarchical approaches can be found in [20]. It is a general planning framework for the synthesis of operating procedures following a top-down methodology. The knowledge representation scheme is a translation of the SP88

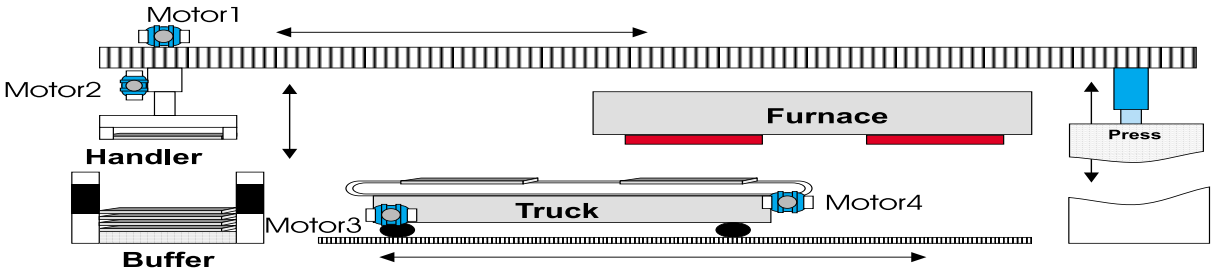


Figure 2. A Manufacturing System.

standard, which allows for creating a *procedural knowledge base* at different abstraction levels. The planner then applies basic HTN [8] techniques in order to find a low level procedure which meets the process of products introduced as problem.

In this system the user has to introduce a great deal of knowledge to solve a problem, and the main role is left to the representation and management of the procedural knowledge base. Therefore, the planner, and thus the autonomy of the approach, is only a very small part of the whole system, and its operation lies on a *static* combination of procedures at different abstraction levels where problems as detection of conflicts, preservation of invariants, and even order relations (between procedure steps) must be hand coded by end users.

However, in order to obtain a efficient and realistic system that applies hierarchical planning techniques, it is necessary to reduce the amount of work that a control engineer has to do in order to describe a manufacturing domain, and in order to find a program that control its operation.

In addition, the control program obtained must have a sufficient level of detail such that it incorporates every necessary action to carry out a correct execution. Present approaches for the synthesis of operating procedures [1, 14] or machining process [16] do not obtain complete and realistic solutions in this sense because plans obtained are operation procedures intended to be executed by human operators, thus they lack of the necessary level of detail to be considered control programs and executed by a computer, or they are only focused in a small part of the overall manufacturing system.

The approach we present mixes hierarchical knowledge and reasoning aspects with POCL techniques in order to reduce the user effort in the domain description and problem solving phases, and also, in order to obtain complete plans so that they can be considered hierarchical control sequences, which can be easily translated into standard representations of control programs.

Next sections describe in detail our approach. Section 3 is devoted to describe how to represent a manufacturing domain as a hierarchy of agents, and how the knowledge about the behavior and properties of agents is inherited between different abstraction levels. In sections 4 and 5 we show the problems and plans representation of our approach. Section 6 introduces the planning algorithm and the remaining sections show the future work and conclusions about this approach.

3 DOMAIN REPRESENTATION

Our planning architecture conceives a plant as a multi-agent domain (see Figures 2 and 3) where every agent represents the knowledge about the relevant properties and behavior of every factory device. Some aspects of the knowledge representation and planning algorithm here presented are based on a previous system (MACHINE [6]), which uses a non-hierarchical *agent centered* domain model for representing a manufacturing system. In MACHINE the behavior of every agent is described as an automaton and every transition of the automaton is represented as a control activity (Figure 4), using an expressive and rich language in order to represent actions as intervals and, in addition, to manage different kinds of conflicts and interferences which may arise in complex domains like manufacturing systems.

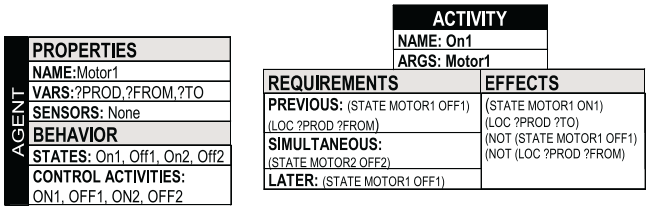


Figure 4. Structure of a primitive Agent.

In this approach, a planning domain is a hierarchy of agents where the root (a "dummy" agent) represents the whole plant (Figure 2), leaf nodes are *primitive* agents corresponding to the field devices of the plant and intermediate nodes are *aggregate* agents, i.e., agents whose structure and behavior are described at higher abstraction levels and which represent a composition of a set of agents at lower levels of abstraction.

Hence, a manufacturing domain is structured at different abstraction levels. Lowest level agents are represented as shown in Figure 4 and its actions are called *primitive activities* intended to be executed by a device.

An aggregate agent is represented as shown in Figure 5. An *aggregate activity* is represented as a primitive one but with an additional property called *expansion*. An example of an aggregate agent and its components can be seen in Figure 6.

The expansion slot of an aggregate activity is used to specify different ways to carry out that activity. These alternative ways are described as a set of different methods [2] where every method is represented by a set of literals (which can be ordered) representing a

[2] Not in the strict sense of HTN.

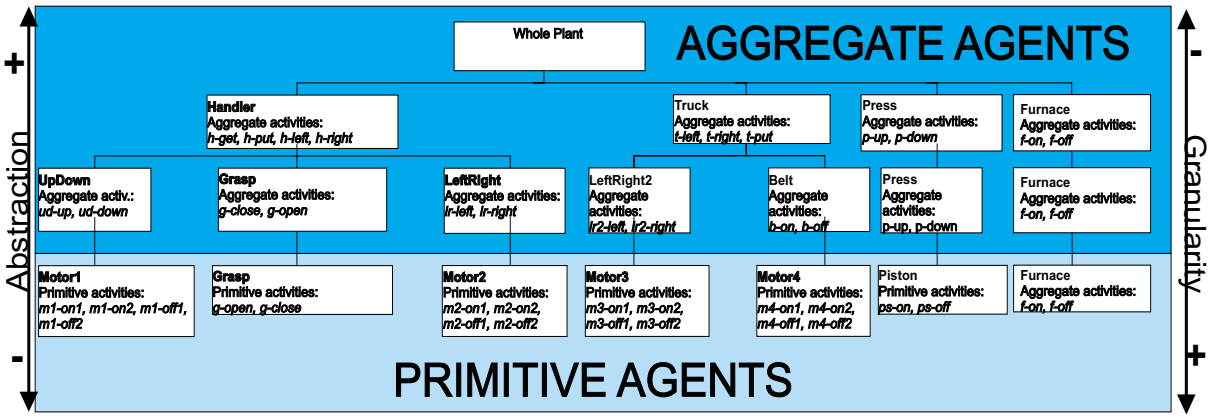


Figure 3. A Hierarchical Domain.

Figure 5. Structure of an Aggregate Agent.

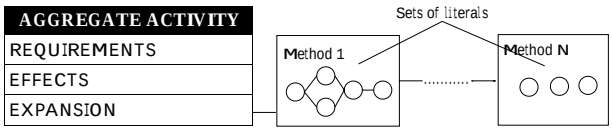


Figure 7. The Expansion Slot of an Aggregate Activity.

problem to be solved by the agents of the next abstraction level (see Figure 7).

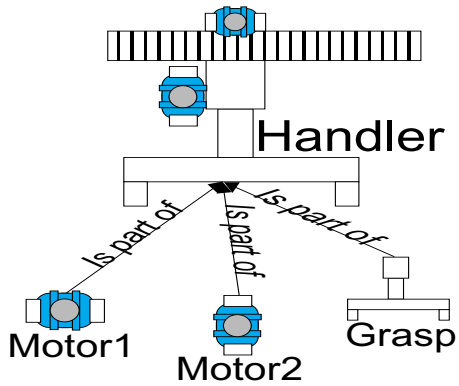


Figure 6. The Handler Aggregate Agent.

In the next section we show how relations between agents and activities at different levels of detail are represented.

3.1 Levels of abstraction and knowledge inheritance

Every aggregate agent of a hierarchical domain is composed of a set of agents at a lower level of abstraction (or higher level of granularity [11]) and it is part of another agent at a higher abstraction level. Hence, properties, activities, states and literals of every agent have an abstraction level associated with them.

Properties and behavior of a given aggregate agent are related with those of its components by means of the *interface* of the aggregate agent. The interface is actually a set of constraints inheritance rules which defines how variables of every component agent inherit their domains and constraints from the ancestor agent (Figure 8).

The aggregate activities of an aggregate agent (its behavior) are represented at a greater *grain size* than the activities of its components. For example, the activities of the agents *UpDown*, *LeftRight* and *Grasp* (see Figure 3) have a lower grain size than the activities of the agent *Handler*. The effects of activity *H-RIGHT* of the agent *Handler* are

(LOC Handler Position2),(STATE Handler Carried).

These literals have a very low level of granularity, and they correspond with the real situation in which the *Handler* agent is up and over the *Truck*, it grasps a piece and it moves to its right. The effects of activity *LR-RIGHT* of the agent *LeftRight* are

(LOC Handler Position2), (LOC LeftRight Position2),(STATE LeftRight Right),

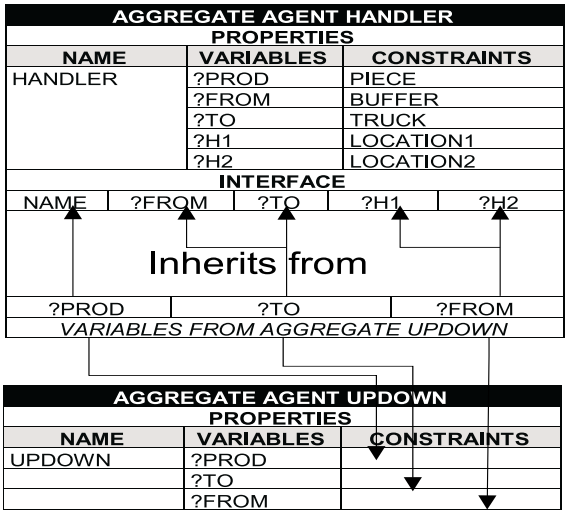


Figure 8. Properties and constraints inheritance.

which have a higher level of granularity and which mean that the *LeftRight* agent is at the same position of the *Handler* agent and, since it is a different agent, its state is moving to its right.

As can be seen, the literals of the requirements and effects of an aggregate activity of an agent may have a different granularity level than the ones of its component agents, and also it is possible that literals of a given granularity level may be different from literals of higher or lower granularity levels.

So, in order to maintain the coherence between different granularity levels activities and literals at different levels must be associated somehow.

The correspondence between a given aggregate activity of an aggregate agent and the activities of its components, and between their different granularity literals, is based on an *activity expansion process* where the aggregate agent's interface and also the expansion slot of the aggregate activity play the key role. The process is described as follows:

- The expansion slot of an aggregate activity of an aggregate agent is represented as a set of methods, where every method is a set of literals at the aggregate granularity level which represents a problem to be solved by the activities of its component agents.
- The literals of these activities have a different granularity, so the activity expansion process applies the rules of the interface of the aggregate agent to every literal in order to articulate the change of granularity. Hence, the interface works as an articulation function [11], between abstraction levels, as follows: every subgoal of a method of an aggregate activity a_g of an aggregate agent g is represented as a literal $(f_1\ x_1\ ...\ x_n)$ where every argument x_i has a domain and constraints at the abstraction level of a_g, then applying the function to this literal will result in a new literal $(f_2\ y_1\ ...\ y_m)$

$$(f_1\ x_1\ ...\ x_n) \xrightarrow{a.f.} (f_2\ y_1\ ...\ y_m)$$

at a higher level of granularity where every argument y_i is a new argument whose domain and constraints must be consistent with the new granularity level. This literal represents a new subgoal to be solved by the activities of the components of g.

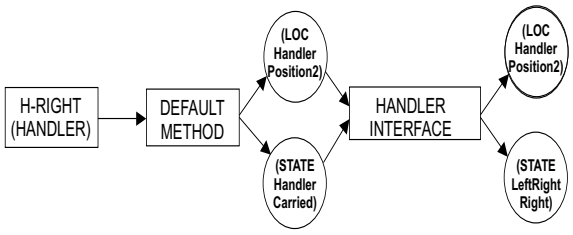


Figure 9. Expansion of Aggregate Activity.

Following this process we obtain a set of literals of lower granularity that represent a problem which must be solved by the activities of the component agents of g by means of a *generative process*.

In addition, the expansion has always at least a *default method* whose literals are the set of effects of the activity. So, a default and *domain independent* expansion process can always be applied for every aggregate activity. Figure 9 shows an example of expansion using a default method. The effects of the activity *H-RIGHT* of the agent *Handler* are the subgoals of the method, and they are mapped by its interface and translated into a set of different literals at a higher level of granularity. These new literals must be solved by the activities of the agents *LeftRigh, UpDown* and *Grasp* at the lower abstraction level.

In addition to this model, the architecture provides a predefined hierarchy of generic classes of aggregate and primitive agents in order to simplify the task of building planning domains, in such a way that every specific agent of the domain is described as an instance of a class of agents (like a *drag & drop* operation).

This model of agents and actions differs of HTN [8] techniques in that expansions (reductions) are not predefined and static substitution rules but a domain independent and dynamic generative process. It also differs from hierarchies of abstraction spaces since sets of operators and literals may be different from an abstraction level to each other. This is because abstraction levels of the proposed hierarchy are based in an increasing semantic granularity instead of in *literal dropping* as in [15, 18].

It possibly looks like models of action in SIPE [21] and OPlan [19], however, their decomposition of actions is defined by the user (plots in SIPE), while in our model the use of an interface between an aggregate and its components and also the expansion of aggregate activities allows for a well defined decomposition by means of the articulation function and default methods, without any participation of the user.

In next sections we will describe the problems and plans representation used in this architecture.

4 PROBLEMS REPRESENTATION

A problem description is a specification of process on products, i.e., a recipe. In our architecture, a problem is represented as an ordered set of literals which represents the process to be carried out by aggregate agents of highest abstraction level. As can be seen at the top of Figure 11 (Step 0) the set of literals received as input by the algorithm represents the ordered set of operations (in SP88 a *procedure*

```
HYBRID (Domain, Level, Agenda, H−Plan)
  If Agenda is Empty
    Then
        If PrimitivePlan? ( H−Plan [Level] )
            Return H−Plan
        Else
          1. RefAlternatives = HowToRefine? (Domain, H−Plan)
          2. While RefAlternatives is not Empty
             2.1. How = Extract (RefAlternatives)
             2.2. REFINE (Domain, How, Level, Agenda, H−Plan)
             2.3. Result = HYBRID (Domain, Level, Agenda, H−Plan)
             2.4. If Result ≠ FAIL
                    Then Return Result
          3. Return FAIL
    Else
        Result = GENERATE (Domain, Level, Agenda, H−Plan)
        If Result = FAIL
            Then Return FAIL
            Else Return HYBRID (Domain, Level, Agenda, H−Plan)
```

Figure 10. The hybrid planning process

recipe) to be carried out by the highest level agents of the manufacturing system represented in Figure 3, that is, a piece must be located in the truck, then it is heated and, finally, it is pressed.

5 PLANS REPRESENTATION

A plan obtained by this architecture is a hierarchy of control sequences (plans) at different levels of granularity, that is, a *hierarchical plan*. Every level in a hierarchical plan is a sequence of control activities to be carried out by agents at the same or higher abstraction level. The last level of the plan is a sequence of primitive control activities (Figure 11).

Every aggregate activity a of an aggregate agent g in a plan level has associated a set of lower level activities of the components of g, which have a causal relation between them and which solves the expansion of a. Hence, a plan can be seen as a modular control program that can be easily translated into standard representations of modular control programs like GRAFCET [12] .

The architecture obtains such a plan following the planning process described in the next section.

6 PLANNING PROCESS

The planning process is a generative and regressive planning algorithm at different levels of detail such that single level plans at higher granularity are refined into lower granularity plans, until no aggregate activities exist on the lowest abstraction level of a hierarchical plan. The generative process is based on a previous non-hierarchical planner [6].

The input to this process is a hierarchical domain and a recipe at the highest abstraction level (a procedure level recipe in SP88, Figure 1). That recipe is preprocessed in order to build a hierarchical plan *H-Plan* with a single abstraction level, containing a set of literals which represent the problem stated by the recipe. As can be seen in Figure 10, the hierarchical domain *Domain*, the initial abstraction level *Level* (the highest one is 1), an initialized task agenda *Agenda* and the initial hierarchical plan *H-Plan* are passed as inputs to the hybrid algorithm. Then it proceeds as follows:

- First, by means of a generative process it obtains a sequence of control activities to be carried out by the highest level agents.
- Second, if the sequence obtained is only composed by primitive activities then the problem is solved. Otherwise, the sequence is *hierarchically refined*, that is, the algorithm expands every aggregate activity, according to its agent interface and its default method or any other method specifically defined, obtaining a new lower level problem.
- Third, the algorithm recursively proceeds to solve the new problem by the agents at the next level.

This is a very general description of the algorithm but, the following describes some important details about the more relevant functions and procedures involved in the algorithm.

HowToRefine?. The result of this function is a list of refinement alternatives of activities which actually represents a heuristic for refining a hierarchical plan *H-Plan*, given a hierarchical domain *Domain*. Depending on the returned heuristic, the behavior of the hybrid algorithm may vary between an ABSTRIPS and an HTN-like behavior. Although many heuristics may exist, the function may return always a default heuristic. This default heuristic is represented by a list with a single element, and its application by the procedure

REFINE results in the expansion of all activities of a given abstraction level, in *H-Plan*, by means of their default method.

REFINE. This procedure applies the above described *activity expansion process* to the aggregate activities of a hierarchical plan *H-Plan*, according to the refinement alternative *How* returned by **Extract**, and taking into account that activities with a lower or equal granularity level than *Level* may be expanded. When **HowToRefine?** returns a unique alternative, representing the default heuristic, it is worthy note that this heuristic applied by **REFINE** turns the **HYBRID** procedure into an *Any Time* hierarchical planning algorithm, because it is able to obtain a plan with no pending subgoals at a given abstraction level (see Figure 11). Additionally, if the refinement process requires it, **REFINE** may introduce a new granularity level in the hierarchical plan, and also may switch to the next abstraction level (increasing *Level*).

GENERATE. This procedure is a generative and regressive planning algorithm based on MACHINE, which is able to represent and reason about actions as intervals and to manage others different kinds of conflicts and interferences which arise in complex domains. However, its features has been extended in order to manage the knowledge inherited by lower level plans from previous abstraction levels in a hierarchical plan (as activities order constraints, established intervals between abstract activities, or the ownership of an activity to the expansion of a more abstract one).

This algorithm solves all of the single-level flaws registered in an agenda *Agenda*, taking into account that, although inherited order relations must be maintained, it is possible to interleave activities belonging to different expansions. These flaws are solved at the next abstraction level of the hierarchical plan. **GENERATE** finally returns a hierarchical plan whith no single-level flaws, but which may contain unexpanded activities.

The algorithm ends when all activities of the lowest abstraction level in *H-Plan* are primitives and there are no pending flaws in *Agenda*. Therefore, the final plan obtained by this algorithm is a hierarchy of control sequences at different granularity levels (See Figure 11 (Step 5)).

As can be seen, the hybrid algorithm here introduced mixes hierarchical and POCL techniques in such a way that the knowledge hierarchy guides the hierarchical reasoning process. Thus, as the hierarchical domain contains a fixed number of abstraction levels, the number of hierarchical refinement levels is also fixed.

6.1 Comparison with other approaches

This approach presents important advantages with respect to previous hierarchical approaches due to the introduction of new issues in the general framework of hierarchical planning [15, 18, 22]. Next we describe the more important ones:

- The default expansion method, defined as the set of effects of an aggregate activity, allows for conceiving the expansion of an activity as a domain independent process. In addition, as it is possible to define alternative expansion methods, the expressiveness of HTN techniques is maintained.
- It is possible to represent a domain as Abstraction Hierarchies [15, 18] and to follow a reasoning process similar to the one used in ABSTRIPS-like approaches. However, the concept of interface and the expansion process here introduced allow for articulating the abstraction levels in a more general way, with a more expressive language and, as we will see, with a reasoning process able to obtain real solutions.
- Unlike HTN [22] techniques, the expansion process of an activity is dynamic and flexible. This means that there not exist a previously fixed reduction of activities, on the contrary, the expansion process of every activity poses a set of lower abstraction goals which have to be dynamically achieved by the agents' activities of the next abstraction level. Thus, the generative process of the hybrid algorithm dynamically stablishes the set of lower level activities that solves the posed problem, in such a way that the expansion process is independent from lower abstraction levels and, therefore, accesible solutions at a given level of the hierarchy are not completely fixed.

In the next section we will discuss some aspects to take into account about the correctness and completeness of this algorithm.

6.2 Correctness and completeness issues

In order to preserve the completeness and correctness of the planning process it is necessary to establish a set of constraints about the way agents and their activities inherit the knowledge of higher abstraction agents. In particular, one of these constraints states that goals with not achieving activity, at a given abstraction level, turn the hierarchical plan unsolvable. Additional completeness constraints are defined in aggregate agents' interfaces and others are "guidelines" on the domain definition in order to represent requirements and effects of activities at different granularity levels. These constraints must be satisfied in the domain elaboration phase of problem solving and must be checked in the action expansion process.

However, although these constraints maintain several established conditions between levels, the Downward Refinement Property (DRP)[15, 22] cannot be satisfied. This property states that the existence of a ground-level solution implies the existence of an abstract-level solution. The contrapositive of this property states that unsolvable conflicts at higher levels always appear in lower ones. Therefore, if this property holds, it is not necessary to refine a plan with an unsolvable conflict. Thus, in this case backtracking between levels is allowed and preserves the algorithm completeness.

Hierarchical planning approaches as [15, 18, 22] are examples about how this property can be satisfied by imposing syntactic constraints in the definition of a domain. However, these constraints reduce the expressiveness of the domain description language.

In abstraction hierarchies [15] levels of abstraction are built from bottom to up by dropping literals from a set of *ground* operators (not reducing the granularity of operators, but relaxing its conditions). In this approach, the set of literals of a given abstraction level contains all literals of the previous level, so lower abstraction levels directly inherit all previous level literals and, hence, every conflict that appears at higher levels also appears in lower levels.

In HTN hierarchies [22], the DRP does not hold in general, but it is possible to establish syntactic constraints in the definition of hierarchical operators in order to satisfy it. However, as can be seen in [22], the syntactic constraints impose that every non-primitive operator has an unique sub-operator which inherits all preconditions and effects of the parent. Therefore, these constraints maintain the grain size of operators between levels.

In our approach, the DRP does not hold because abstraction levels have an increasing size of grain. The states of the automaton that describes the behavior of an aggregate agent are not directly inherited by its components, because they are different states at different

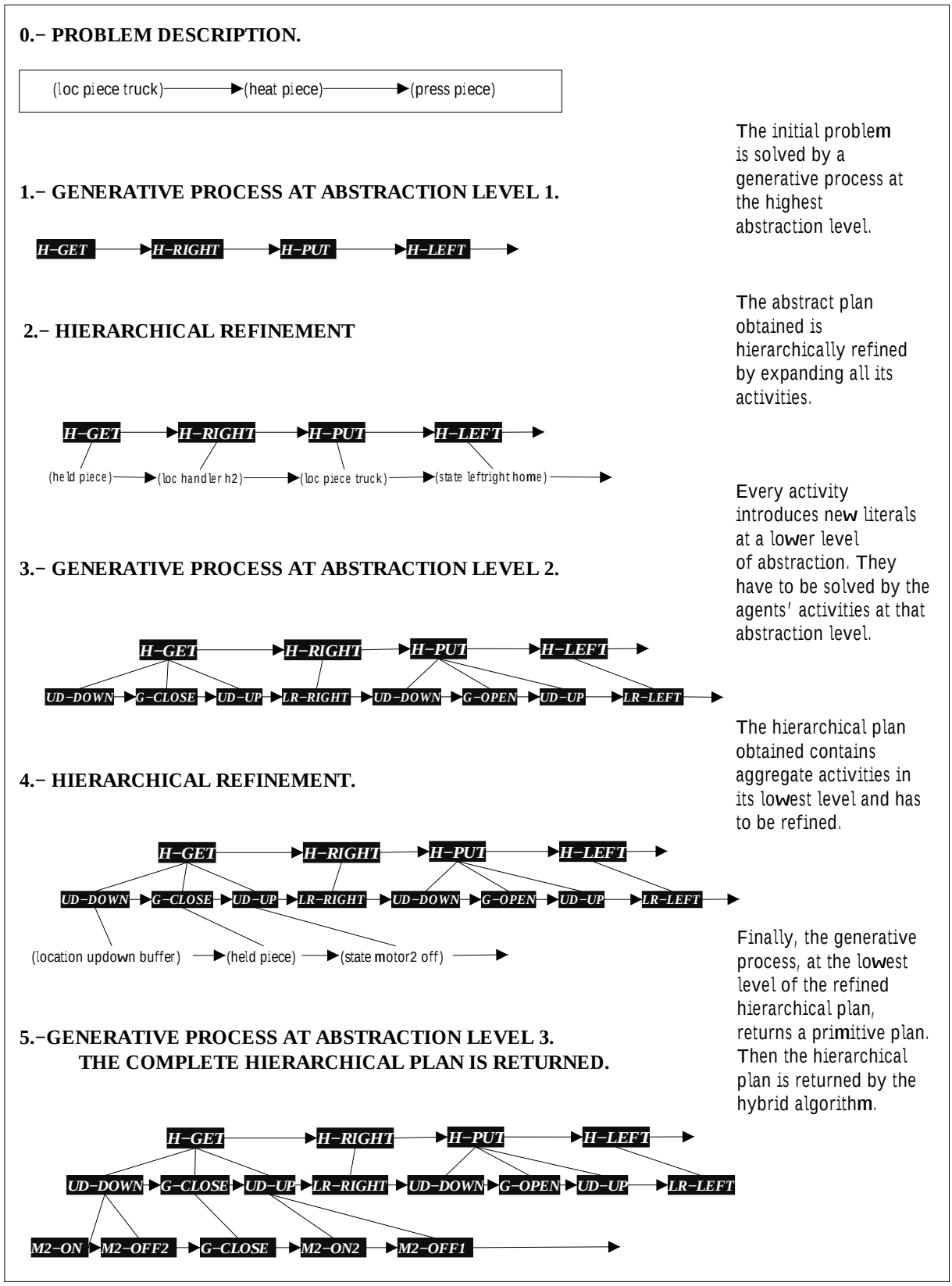


Figure 11. A hierarchical plan obtained by **HYBRID** applying the default refinement heuristic.

granularity levels. This means that there exist literals established in higher levels that disappear in lower levels, so unsolvable conflicts in higher levels may not be inherited by lower levels.

Therefore, we have to face new problems which arise with the application of hybrid planning techniques in a domain representation based on granularity levels:

1. If the DRP does not hold, the algorithm is forced to refine a plan with an unsolvable conflict. The refinement stops when the planner discovers, at a lower level, that the conflict is solved or when the conflict is definitely unsolvable at the lowest level.
2. The activities involved in an unsolvable conflict have to be expanded in order to test the conflict at a lower level. This means that a plan may contain activities and literals at different levels of abstraction (or granularity)
3. The time efficiency of the planner may be reduced if it expands activities up to the lowest level every time that an unsolvable conflict arises.

The second problem has an inmediate solution because the proposed domain representation allows for several levels of abstraction in a plan. For every literal and activity, an unique associated abstraction level always exists, so the harmful effects of the *hierarchical promiscuity* [21] are avoided.

However, a solution to the first and third problem is complex and may be found by extending the heuristics and plans representation currently used in order to manage heterogeneous plans and to offer a correct solution in a reasonable time. At present we are working in this direction but, as can be seen in the next section, the results presented, comparing to the system this approach it is based on, are very promising.

7 EXAMPLES

This section shows the performance of this architecture with respect to the non-hierarchical planner it is based (MACHINE) in the solving of two manufacturing problems.

The layout of the first problem is shown in Figure 12. The problem in this toy plant consists in carry out water from TANK1 to TANK3 and acid from TANK2 to TANK4, but taking into account that they cannot be mixed and that there is only one pump. The plant is represented at two abstraction levels, the circuits are represented in the highest level and the valves in the lowest one. The hierarchical plan obtained is shown in Figure 13 and the performance of the hybrid planning process is shown in Figure 15.

The batch plant of Figure 14 is the configuration of the second problem. In this batch problem there are three types of raw products: an ingredient A, stored into tank T-501, an ingredient B, stored into tank T-505 placed somewhere out of the system, and an ingredient C stored into tank T-504 also out of the system. The hierarchical domain is represented at two granularity levels. The agents of the highest level are represented by aggregating the properties and the behavior of lowest level agents as can be seen in Figure 14

The manufacturing problem *for the lowest hierarchical level* is defined by the following sequence of transformations:

1. STEP 1. Add ingredient B to ingredient A in reactor R-501. During this operation, the mixture must be in agitation.
2. STEP 2. Heat the mixture.
3. STEP 3. Add ingredient C to the mixture maintaining the agitation. During this mixing operation a residual gas is generated which must be evacuated through the scrubber S-501. Part of this gas is condensed and it precipitates at the bottom of S-501. Once the mixing operation ends, this residual liquid must be carried into the external tank T-503.
4. After the addition of ingredient C, the mixture must be cooled and carried into tank T-502.

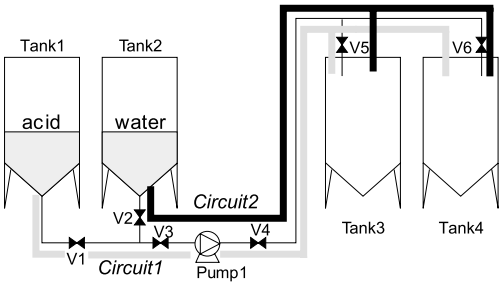


Figure 12. Layout of Problem1.

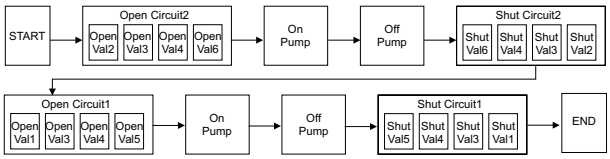


Figure 13. A hierarchical plan for Problem1.

This problem definition is a control recipe specified at phase level. In this case, MACHINE solves the problem exploring about 6000 nodes. However, with this new hierarchical approach the problem may be described as a recipe at operation level whit only one operation: Mix the ingredients in REACTOR. Then the hybrid planning process will obtain first a phase level recipe and, afterwards, the set of control activities of lowest level agents.

These examples show the usefulness of this approach from the

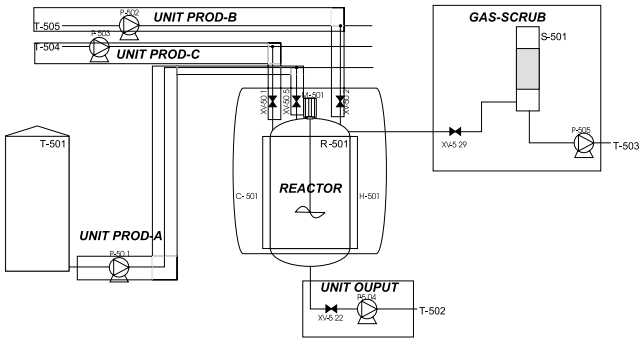


Figure 14. Layout of Problem2.

	EXPLORED NODES	
	MACHINE	HYBRID ARCHITECTURE
PROBLEM 1	400	200
PROBLEM 2	6000	800

Figure 15. Compared search results for Problem1 and Problem2.

view point of a control engineer and its expressiveness and search complexity benefit (see Figure 15). In the next section we describe how to extend this architecture in order to obtain truly realistic solutions.

8 FUTURE WORK

This approach is a step forward in the building of an architecture able to face real world control problems and to obtain fully applicable solutions. However, there are still some steps which must be faced to reach that goal. These steps come up closely related with some of the features that control programs are expected to have, such as robustness or safety.

In a factory, sensory information is a source of uncertainty about the state of the system which forces control engineers to introduce decision and alternative operation steps in a control program, according to the different states of a given sensor. The knowledge representation presented here allows for describing sensors and sensory information but the planning algorithm must still be extended in order to manage the "a priori" uncertainty about states of sensors. This means that the final result of the planning process should be a modular control program with conditional decision structures[9, 17].

This implies that the underlying search space will grow exponentially due to the combinatorial complexity induced by the introduction of conditional branches in a plan. However, it must be said that the effect of this combinatorial explosion can be reduced since the scope of conditional branches can be focused, or isolated, on the basis of a top-down hierarchical process like the one described in this work.

Finally, this approach is being developed within the framework of assisted development of control programs. This means that human operators could interact with the planner and impose their decisions at certain points during the search in a mixed-initiative planning process. The reason for such an interaction is that, in many real world problems, the vast amount of knowledge required for obtaining a solution would produce unrealistically large planning domains and this knowledge can not be completely included. Therefore a planning process must always be open to a possible human interaction which could provide that missing knowledge, what could be seen as a control heuristic to guide the search or to obtain optimal solutions.

The approach presented here follows this direction and it intends to approach these problems in the near future.

9 CONCLUSIONS

We have presented a hybrid architecture which mixes hierarchical planning and POCL techniques, in order to build modular and hierarchical control programs for manufacturing systems.

This architecture is based on a hierarchy of agents by levels of abstraction in such a way that the information granularity of agents, literals and actions increases as the level of abstraction decreases (Figure 3). This representation leads to define different alternatives to existing abstract plan refinement techniques (reduction methods in HTN or plan refinement in ABSTRIPS), the activity expansion process presented is one of them.

The application of hierarchical problem solving techniques results in a lower time and space complexity of this architecture with respect to the system it is based on. However, though the hierarchical domain representation model and planning can reduce the benefit of using these techniques in some cases [2, 3, 4, 10], it has clear advantages from the point of view of computer aided design of control programs.

On the one hand, this approach provides an easy entry-level for end users (control engineers). The hierarchy of agents of a domain accepts SP88 standard descriptions, usually handled by control engineers, so the knowlege can be introduced painlessly.

On the other hand, plans are designed and represented following a top-down process which makes them easier to understand by a human user.

In conclusion, this hierarchical representation and planning process provides a greater efficiency with respect to the non-hierarchical previous version, but, and this is more important for a real world planner, it closes the gap between the planner and their end-users providing a higher degree of integration with them by means of a friendly input level for incorporating knowledge and a more understandable output level.

ACKNOWLEDGEMENTS

This work has been supported by C.I.C.Y.T. under project TAP99-0535-C02-01.

References

[1] R. Aylett, G. Petley, P. Chung, J. Soutter, and A. Rushton, 'Planning and chemical plan operation procedure synthesis: a case study', in *Fourth european conference on planning*, pp. 41–53, (1997).
[2] F. Bacchus and Q. Yang, 'Downward refinement and the efficiency of hierarchical problem solving', *Artificial Intelligence*, **71**, 43–100, (1994).
[3] C. Backstrom and P. Jonsson, 'Planning with abstraction hierarchies can be exponentially less efficient', in *Proc. of IJCAI 95*, pp. 1599–1604, (1995).
[4] R. Bergman and W. Wilke, 'Building and refining abstract planning cases by change of representation language', *JAIR*, **3**, 53–118, (1995).
[5] *PLANET News (Issue No.1)*, eds., S. Biundo and B. Schattenberg, PLANET: European Network on Excellence in AI Planning, 2000.
[6] L. Castillo, J. Fdez-Olivares, and A. González, 'Automatic generation of control sequences for manufacturing systems based on nonlinear planning techniques', *Artificial Intelligence in Engineering*, **4**(1), 15–30, (2000).
[7] L. Castillo, J. Fdez-Olivares, and A. González, 'A three-level knowledge based system for the generation of live and safe petri nets for manufacturing systems', *To appear in Journal of Intelligent Manufacturing*, (2000).
[8] K. Erol, J. Hendler, and D. Nau, 'UMCP: A sound and complete procedure for hierarchical task-network planning', in *AIPS-94*, (1994).
[9] O. Etzioni, S. Hanks, D. Weld, D. Draper, N. Lesh, and M. Williamson, 'An approach to planning with incomplete information', in *Proc. Third. Int. Conf. on Principles of KRR-92*, pp. 115–125, (1992).
[10] F. Giunchiglia, 'Using abstrips abstractions – where do we stand?', *Artificial Intelligence Review*, **13**, 201–213, (1999).
[11] J. Hobbs, 'Granularity', in *IJCAI 85*, pp. 432–435, (1985).
[12] IEC, 'Preparation of function charts for control systems', Technical Report IEC-60848, International Electrotechnical Commission, (1988).
[13] Instrument Society of America (ISA), *Batch control Part 1: models and terminology (SP-88)*, 1995.
[14] I. Klein, P. Jonsson, and C. Backstrom, 'Efficient planning for a miniature assembly line', *Artificial Intelligence in Engineering*, **13**(1), 69–81, (1998).

[15] C. A. Knoblock, *Generating Abstraction Hierarchies*, Kluwer Academic Publishers, 1993.
[16] D. Nau, S. K. Gupta, and W. C. Regli, 'AI planning versus manufacturing-operation planning: A case study', in *IJCAI-95*, pp. 1670–1676, (1995).
[17] L. Pryor and G. Collins, 'Planning for contingencies: A decision based approach', *Journal of Artificial Intelligence Research*, **4**, 287–339, (1996).
[18] D. E. Sacerdoti, 'Planning in a hierarchy of abstraction spaces', *Artificial Intelligence*, **5**, 115–135, (1974).
[19] A. Tate, B. Drabble, and R. Kirby, 'O-PLAN2: An open architecture for command, planning and control', in *Intelligent scheduling*, eds., M. Zweben and M. Fox, Morgan Kaufmann, (1994).
[20] S. Viswanathan, C. Johnsson, R. Srinvivasan, V. Venkatasubramanian, and K.E. Ärzen, 'Automating operating procedure synthesis for batch processes. part I: Knowledge representation and planning framework.', *Computers and Chemical Engineering*, **22**(11), 1673–1685, (1998).
[21] D. E. Wilkins, *Practical planning: Extending the classical AI planning paradigm*, Morgan Kaufmann, 1988.
[22] Q. Yang, *Intelligent Planning. A decomposition and Abstraction Based Approach*, Springer Verlag, 1997.

Structure and Complexity in Planning with Unary Operators

Carmel Domshlak and **Ronen I. Brafman**[1]

Abstract. In this paper we study the complexity of STRIPS planning when operators have a single effect. In particular, we show how the structure of the domain's causal graph influences the complexity of planning. Causal graphs relate between preconditions and effects of domain operators. They were introduced by Williams and Nayak, who studied unary operator domains because of their direct applicability to the control of NASA's Deep-Space One spacecraft. Williams and Nayak's reactive planner can be trivially extended into a polynomial time plan generator in the context of tree-structured causal graphs. In this paper, we treat more complex causal graph structures, such as undirected polytrees, singly-connected networks, and general DAGs. We show that a polynomial time plan generation algorithm exists for graphs that induce an undirected polytree. More generally, we show that a certain relation exists between the number of paths in the causal graph and the complexity of planning in the associated domain.

1 INTRODUCTION

Generating plans in the context of the STRIPS representation language is known to be a difficult (P-SPACE complete) problem [5]. Thus, various authors have explored the existence of more constrained problem classes for which planning is easier. For example, Bylander showed that STRIPS planning in domains where each operator is restricted to have positive preconditions and one postcondition only is tractable. Bäckström and Klein [1] considered other types of local restrictions, but using a more refined model in which two types of preconditions are considered: *prevail* conditions, which are variable values that are required prior to the execution of the operator and are not affected by the operator, and *preconditions*, which *are* affected by the operator. For example, [1] have shown that when operators have a single effect, no two operators have the same effect, and each variable can be affected only in one context (of prevail conditions) then the planning problem can be solved in polynomial time. However, these restrictions are very strict, and it is difficult to find reasonable domains satisfying them.

More recently, Williams and Nayak [3] studied planning problems where all operators affect a single variable, in the context of their work on controlling NASA's Deep-Space One spacecraft. In this context, they defined the notion of a *causal graph* which relates the causal structure of the domain, i.e., how different variables play a role in our ability to affect other variables. A causal graph is a directed graph whose nodes are the domain propositions. An edge (p, q) appears in the causal graph if some operator that changes the value of q has a prevail condition involving p. When the causal graph is a tree, it is easy to determine a serializability ordering over any set of sub-goals, and consequently, obtain a plan in polynomial time.

An important byproduct of Williams and Nayak's work is its demonstration that unary operator domain are of practical interest. Interestingly, unary operator domains show up naturally in another application – answering dominance queries in CP-networks [4].

Our work continues Williams and Nayak's study of unary operator domains, concentrating on the relationship between the domain's causal graph and the complexity of plan generation and plan existence. In particular we prove the following results:

- When the undirected graph induced by the causal graph is singly connected, plan existence and plan generation can be performed in $O(e)$ time (where e is the number of edges in the causal graph).
- When the causal graph is singly connected, plan generation is in NP.
- When the causal graph has more than three paths between two variables, plan generation is NP-hard.
- In general, the complexity of plan generation can be bound by a function of the number of paths within the causal graph.

The rest of this paper is devoted to a more formal presentation of these results and their proofs.

2 COMPLEXITY RESULTS

We now show how, by bounding the structural complexity of the causal graph, we can bound the complexity of plan generation. Recall that we use a propositional language to describe the state of the world, and that our operators are described by a set of prevail conditions – i.e., a set of literals that must hold in a world for the operator to be applicable, a single precondition, and a single post-condition (or effect). The precondition and the post-condition are represented by single literals, one the negation of the other.

2.1 Undirected Polytrees

A polytree is a singly connected graph, i.e., a graph in which there is a single path between two nodes. Here, we consider the case of a causal graph in which there is a single path between every pair of nodes in the induced *undirected* graph. For this class of problems we will present a polynomial time planning algorithm. We will rely on [6]'s formulation of the POP algorithm, and we will assume that the reader is familiar with that algorithm.

Our algorithm proceeds in two stages: First, we perform a forward check step. Following this step, which takes time linear in the size of the input, we can answer the question whether or not a plan exists. If the answer if positive, we run a particular instantiation of the POP

[1] Dept. of Computer Science, Ben-Gurion University of the Negev, P. O. Box 653, Beer-Sheva 84105, Israel, e-mail: {dcarmel, brafman}@cs.bgu.ac.il

algorithm which generates the plan without backtracking in linear time.

The forward checking procedures, described in Figure 1, works as follows: we perform a topological sort of the causal graph and start processing each node (=variable) from top to bottom. At each point, a set of operators is associated with each variable, i.e., the set of operators that can change the value of that variable. Initially, this would be the set of all such operators. Now, for each variable v, we check whether its value in the initial state, v^0, differs from its value in the goal state, v^*. If this is the case and there is no operator transforming v^0 to v^* we return *failure*. If $v^0 = v^*$ then we first check whether there are two operators associated with v that have both its values as effects. If this is the case, then, intuitively, this implies that we can change v's current value and still regain the value needed in the goal state. Otherwise, we mark v *locked* and we extract all operators in which the negation of v's current value appears as a prevail condition – it is clear that we will never be able to apply these operators in a valid plan.

It is apparent that the procedure Forward-Check's running time is linear in the number of operators.[2]

Procedure Forward-Check (Π, Λ, $\mathcal{G}$)

1. Topologically sort all variables $\mathcal{V}$ based on the the causal graph.
2. For each variable $v \in \mathcal{V}$, call Recursive-Locking(Π, Λ, v, $\mathcal{G}$), respecting the above ordering.
3. If all calls to Recursive-Locking return success, then return success. Otherwise return failure.

Procedure Recursive-Locking(Π, , Λ, v_i, $\mathcal{G}$)

1. If $v_i^0 \neq v_i^*$ then
   (a) If no operator A_i^+ in Λ has v_i^* as post-condition, return failure.
   (b) Otherwise return success.
2. If $v_i^0 = v_i^*$ then
   (a) If there are two operators A_i^- and A_i^+ in Λ, that have $\neg v_i^*$ and v_i^* as their post-conditions respectively, return success.
   (b) Otherwise, mark the variable v_i *locked* and remove from Λ all operators that have $\neg v_i^*$ as a precondition or prevail condition. (Note that this requires considering operators affecting the children of v_i only.)

Figure 1. Forward checking procedure

Lemma 1 *Forward-Check returns* success *if and only if a plan exists.*

Clearly, if Forward-Check fails, then no plan exists. To prove the opposite direction we proceed as follows: We define a partial order planning algorithm POP-UPC (partial-order planner for undirected polytree causal graph) and show that it will succeed without backtracking if Forward-Check succeeds. POP-UPC is described in detail in Figure 2. In its description, we assume that a minimal number of operators exists, i.e., if we remove a single operator from the domain, Forward-Check would no longer return success.

Intuitively, POP-UPC works as follows: it maintains a goal agenda sorted based on the causal graph structure: parent variables appear after their descendents. At each point, the next agenda item is selected;

[2] This assumes some appropriate indexing is used. This indexing should be performed once for each planning domain.

Algorithm: POP-UPC ($\langle \mathcal{A}, \mathcal{O}, \mathcal{L} \rangle$, `agenda`, Λ)

1. **Termination:** If `agenda` is empty, return $\langle \mathcal{A}, \mathcal{O}, \mathcal{L} \rangle$
2. **Goal selection:** Let $\langle \vartheta_i, A_{need} \rangle$ be a *rightmost* pair on the `agenda` (by definition, $A_{need} \in \mathcal{A}$ and ϑ_i is one of the preconditions of A_{need}).
3. **Operator selection:**
   (a) If $v_i^0 = v_i^*$ and $\vartheta_i = v_i^*$:
      i. If there are no items on the agenda requiring $\neg v_i^*$ and $A_i^- \notin \mathcal{A}'$, or if $A_{need} = A_i^-$ then $A_{add} = A_i^0$.
      ii. Otherwise, $A_{add} = A_i^+$.
   (b) If $v_i^0 = v_i^*$ and $\vartheta_i \neq v_i^*$ then $A_{add} = A_i^-$.
   (c) If $v_i^0 \neq v_i^*$ and $\vartheta_i \neq v_i^*$ then $A_{add} = A_i^0$.
   (d) If $v_i^0 \neq v_i^*$ and $\vartheta_i = v_i^*$ then $A_{add} = A_i^+$.
4. **Plan updating:** Let $\mathcal{L}' = \mathcal{L} \cup \{A_{add} \stackrel{\vartheta_i}{\rightarrow} A_{need}\}$, and let $\mathcal{O}' = \mathcal{O} \cup \{A_{add} < A_{need}\}$. If A_{add} is newly instantiated, then $\mathcal{A}' = \mathcal{A} \cup \{A_{add}\}$ and $\mathcal{O}' = \mathcal{O} \cup \{A_i^0 < A_{add} < A_i^*\}$ (otherwise let $\mathcal{A}' = \mathcal{A}$ and $\mathcal{O}' = \mathcal{O}$).
5. **Update goal set:** Let `agenda`$'$ = `agenda` - $\{\langle \vartheta_i, A_{need} \rangle\}$. If A_{add} is newly instantiated, then for each of its precondition Q, add $\langle Q, A_{add} \rangle$ to `agenda`$'$.
6. **Threat prevention:** If $A_{add} = A_i^+$, then, for each $A \in \mathcal{A}'$, s.t. $\neg v_i^*$ belongs to the preconditions of A add $\{A < A_{add}\}$ to $\mathcal{O}'$.
7. **Recursive invocation:** POP-UPC ($\langle \mathcal{A}', \mathcal{O}', \mathcal{L}' \rangle$, `agenda`$'$, Λ), where `agenda` is topologically ordered (based on the causal graph with respect to the precondition part of each pair).

Figure 2. POP-UPC algorithm

if it requires achieving some value for v that differs from its initial value, we add an operator to the plan with the desired effect. Otherwise, we need the same value v_* for v as that which appears in the initial state. If no operator was added which has the opposite of v_0 as a prevail condition, we will use the initial state (or in POP terminology, the operator A^0) to achieve this value (i.e., we simply do not change this value throughout the plan). If we added an operator which negates v_0, we must re-establish it, and we add an operator with that effect. No threats arise in POP-UPC, and the ordering constraints are consistent.

Lemma 2 *If Forward-Check was successful then POP-UPC will return a valid plan.*

Proof The Lemma will follow from the following claims:

1. For every agenda item, there exists an operator that has it as an effect.
2. There are no threats in the output of POP-UPC.
3. The ordering constraints in $\mathcal{O}$ are consistent.

(1) The first claim follows from the success of the Forward-Check procedure. It implies that for every variable v if v's initial value differs from its final value, there is an operator for achieving that value. For any other variable, we can always use the initial state as the source of its value. If v's initial and final value are the same and there are no two operators that can change v's value in both directions, then because of the locking mechanism, we will not allow any operator that relies on the value of v that differs from its initial value. Hence, the need for an appropriate precondition will not arise.

(2) Suppose that some operator A_t threatens $A_p \stackrel{\vartheta_i}{\rightarrow} A_c$, i.e.,

- $\mathcal{O} \cup \{A_p < A_t < A_c\}$ is consistent, and
- A_t has $\neg\vartheta_i$ as an effect.

For a given variable v_i, only three operators can have an effect pertaining to v_i: A_0, A_i^+, and A_i^-. POP-UPC forces these operators to be ordered as follows: $A_0 < A_i^- < A_i^+$, so A_c can only be an operator with ϑ_i as a prevail condition. There are two cases to consider: $A_p = A_0, A_t = A_i^-$ and $A_p = A_i^-, A_t = A_i^+$.
Suppose that $A_p = A_0, A_t = A_i^-$. In that case $\vartheta_i = v_i^*$, but the only A_c for which A_0 supplies v_i^* is A_i^-.
Suppose that $A_p = A_i^-, A_t = A_i^+$. In that case, $\vartheta_i \neq v_i^*$, and step 6 guarantees that $A_c < A_i^+$. Hence, again, no threat occurs.

(3) The ordering constraints are consistent if no two operators A_i and A_j are such that $\mathcal{O}$ implies $\{\{A_i < A_j\}, \{A_i < A_j\}\}$. In what follows, A_i will be used to denote an arbitrary operator affecting variable v_i.
First note that each ordering constraint added in Step 4 or Step 6 is between operators affecting a variable and its child (with respect to the causal graph). In particular, if $A_i < A_j$ was added in Step 4 then v_i is a parent of v_j, whereas if $A_i < A_j$ was added in Step 6, v_j is a parent of v_i. In particular, this means that if $A_i < A_j$ is implied by $\mathcal{O}$ then there is a path between v_i and v_j in the undirected graph induced by the causal graph.
Assume, to the contrary that $\mathcal{O}$ implies $A_i < A_j$ and $A_j < A_i$. From the argument above, we know that there is a path between v_i and v_j in the undirected graph induced by the causal graph. By our structural assumption, we know that there is a unique path between v_i and v_j. Thus, the situation is as follows: We have a chain of operators $A_i = A_{i_0} < A_{i_1} < \cdots < A_{i_m} = A_j$ implying $A_i < A_j$, and a chain $A_i = A'_{i_0} > A'_{i_1} > \cdots > A'_{i_{m-1}} > A'_{i_m} = A_j$ implying $A_i > A_j$. Without loss of generality, the internal A'_{i_l} and A_{i_l} are different (otherwise, we can reduce the chain and deduce $A_i < A_{i_l}$ and $A_i > A_{i_l}$).
We know that $A'_{i_1} < A_{i_0} < A_{i_1}$:
(1) If v_{i_0} is a parent of v_{i_1} then $A'_{i_1} < A_{i_0}$ can only stem from Step 6 because $\neg v^*_{i_0}$ is a precondition of A'_{i_1} and $A_{i_0} = A^+_{i_0}$. $A^+_{i_0} < A_{i_1}$ can only stem from Step 4 because $v^*_{i_0}$ is a precondition of A_{i_1}. Hence, A_{i_1} and A'_{i_1} must be different operators. Given the algorithm we must have $A'_{i_1} = A^-_{i_1}$ and $A_{i_1} = A^+_{i_1}$. (Otherwise, we have both $A^+_{i_0} < A^+_{i_1}$ and $A^+_{i_1} < A^+_{i_0}$ which implies conflicting preconditions for $A^+_{i_1}$.)
(2) If v_{i_1} is a parent of v_{i_0} then $A_{i_0} < A_{i_1}$ can only stem from Step 6, $A_{i_1} = A^+_{i_1}$, and $\neg v^*_{i_1}$ is a precondition of A_{i_0}. In that case A'_{i_1} must be $A^-_{i_1}$ and, again A_{i_1} and A'_{i_1} are different.
Continuing with the next variable, v_{i_2} we know that $A'_{i_2} < A^-_{i_1}$ and $A^+_{i_1} < A_{i_2}$. We claim that $A'_{i_2} \neq A_{i_2}$. More specifically, we claim that $A'_{i_2} = A^-_{i_2}$ and $A_{i_2} = A^+_{i_2}$.
First, suppose that v_{i_1} is the parent of v_{i_2}. In that case, $A'_{i_2} < A^-_{i_1}$ is impossible. Hence, v_{i_2} must be the parent of v_{i_1}. From $A^+_{i_1} < A_{i_2}$ we can deduce that $A_{i_2} = A^+_{i_2}$. As in the previous case, the fact that $A'_{i_2} \neq A^+_{i_2}$ follows easily, and hence $A'_{i_2} = A^-_{i_2}$.
Having established that $A'_{i_2} = A^-_{i_2}$ and that $A_{i_2} = A^+_{i_2}$, it is apparent that an inductive argument will allow us to show that for all $n > 0$ we have that $A'_{i_n} = A^-_{i_n}$ and $A_{i_n} = A^+_{i_n}$. This contradicts our assumption that $A_{i_m} = A'_{i_m}$.
■

2.2 Polytrees

In this section we provide an upper bound on the complexity of plan generation when the causal graph is a polytree. In particular, we show that this problem is in NP. However, the question of the exact placement of this problem in the computational complexity hierarchy is left open.

First we make the following observation, upon which we base our proof. The central claim will follow using an induction on the number of variables.

Consider an arbitrary planning problem instance Π with a variable set $\mathcal{V}$, and an operator set Λ. Denote by *Must*(v) the maximal number of times that a variable v *must* change its value in the course of execution of a valid plan for this problem. For the type of problems we deal with, for all variables in $\mathcal{V}$ *Must*(v) satisfies:

$$\textit{Must}(v) \leq 1 + \sum_{Sons(v)} \textit{Must}(u) \tag{1}$$

where *Sons*(v) denote the immediate successors of v in the corresponding causal graph. That is, a variable must change its value at most once for each requested change of its successors (in order to satisfy necessary prevail conditions), and then at most once in order to accept the value requested by the goal state.

Let *MinPlanSize*(Π) denote the size of the minimal plan for the problem Π. Using the *Must* property of the state variables, the following upper bound for *MinPlanSize*(Π) is straightforward:

$$\textit{MinPlanSize}(\Pi) \leq \sum_{v \in \mathcal{V}} \textit{Must}(v) \tag{2}$$

This bound holds for all unary operator domains whose causal graph is acyclic. We will use this bound to prove the following lemma.

Lemma 3 *Plan generation for propositional planning problems in domains whose underlying causal graph is a polytree is in* NP.

Proof In order to prove this claim it is sufficient to show that for any solvable problem instance $\Pi = \langle \mathcal{V}, \Lambda, Init, Goal \rangle$ for domains whose causal graph is a polytree, the length of the minimal (optimal) solution will be polynomial in the size of input. Since the verification of the solution takes time linear in its length, the bound follows. We will show that the length of the minimal solution is less than or equal to n^2, where n is the number of variables in $\mathcal{V}$. Our proof does not rely on the particular initial or goal state, and so we will ignore them, from now on.
The proof is by induction on n, and is based on the previously achieved upper bound on *MinPlanSize*. For $n = 1$, Equation 1 implies that

$$\sum_{v \in \mathcal{V}} \textit{Must}(v) \leq \textit{Must}(v_1) \leq 1 = 1^2.$$

Now, suppose that when $|\mathcal{V}| = n - 1$ then

$$\sum_{v \in \mathcal{V}} \textit{Must}(v) \leq (n-1)^2.$$

Let Π' be some problem instance for which $|\mathcal{V}'| = n$. Suppose that the variables in $\mathcal{V}' = \{v_1, \ldots v_n\}$ are topologically ordered based on the domain's causal graph. Clearly, v_n is a leaf node (i.e., *Sons* $(v_n) = \emptyset$). We will denote by Π the problem instance obtained by removing v_n from the domain, and the corresponding variable set by $\mathcal{V}$. According to Equation 1, for each immediate predecessor v of v_n in the causal graph,

$$\textit{newMust}(v) \leq \textit{Must}(v) + \textit{newMust}(v_n) \leq \textit{Must}(v) + 1$$

where *newMust*(v) denotes *Must*(v) with respect to Π'. Generally, for each variable $v \in \mathcal{V}$,

$$\textit{newMust}(v) \leq \begin{cases} \textit{Must}(v)+1, & if\ there\ is\ a\ path\ from\ v\ \text{to}\ v_n \\ \textit{Must}(v), & otherwise \end{cases} \tag{3}$$

Now,

$$\sum_{v\in\mathcal{V}'} \textit{newMust}(v) \;\leq\; n + \sum_{v\in\mathcal{V}} \textit{Must}(v) \;\leq\; n + (n-1)^2 \;<\; n^2$$

and thus, according to the upper bound on *MinPlanSize*,

$$\textit{MinPlanSize}(\Pi') \leq \sum_{v\in\mathcal{V}'} \textit{newMust}(v) \leq n^2$$

■

Lemma 3 shows that any propositional, "polytree-structured" planning problem, is in NP. Moreover, the size of the minimal solution is bounded by low polynomial in $|\mathcal{V}|$, which does not depend on the size of the whole input, $|\mathcal{V}| + |\Lambda|$. Following subsection will highlight the significance of structural properties in the unary operator planning problems.

2.3 General DAGs

The polytree structure of the causal graph turns out to be crucial for guaranteeing reasonable solution times. As we now show, there are solvable propositional planning problems with an arbitrary acyclic (DAG) causal graph that have minimal solutions of exponential size. Analysis of this class of problems points to the reason for such inherent intractability. This allows us to characterize an important parameter of the causal graph affecting planning complexity and to extend the class of problems which are in NP. However, we also show that most of these restricted problems are NP-complete.

Lemma 4 *Plan generation for STRIPS planning problems with a unary operator domain whose causal graph is acyclic is inherently intractable.*

Proof We prove this claim simply by showing the supporting example. However, we will perform more detailed analysis, postponing the example to the end of the proof. Our analysis is based on the fact that the upper bound for *MinPlanSize*, presented in Equation 2 can be exponential in the size of input. First, we prove this claim, then we show by example that this upper bound can be achieved.
The escalation of the complexity, when the number of variables in $\mathcal{V}$ grows, can be shown by bounding *newMust*(v) using a different method than that of Equation 3. For the problems considered in Lemma 4,

$$\textit{newMust}(v) \leq \textit{Must}(v) + \rho_{v\rightsquigarrow v_n}$$

where $\rho_{v_i\rightsquigarrow v_j}$ denotes the total number of different, not necessary disjoint, paths from v_i to v_j. This means that for a given propositional planning problem with acyclic causal graph, with variables numbered according to a topological sort induced by the causal graph,

$$\textit{Must}(v_i) \leq \sum_{j=i+1}^{n} \rho_{v_i\rightsquigarrow v_j} \tag{4}$$

Thus, the upper bound for *MinPlanSize*, presented in Equation 2 can be exponential in the size of the problem description.

Now we show an example, for which such an exponential upper bound can be achieved. This particular example was used in different context by Bäckström and Nebel in [2]. Consider a propositional planning problem with $|\mathcal{V}| = n$, and *Parents*$(v_i) = \{v_1, \ldots v_{i-1}\}$ for $1 \leq i \leq n$. The operator set Λ consist of $2n$ operators $\{A_1, A_1', \ldots A_n, A_n'\}$ where

$$\begin{aligned} &\textit{pre}(A_i) = \textit{post}(A_i') = 0 \\ &\textit{post}(A_i') = \textit{pre}(A_i) = 1 \\ &\textit{prv}(A_i)[j] = \textit{prv}(A_i')[j] = \begin{cases} 0 & \text{if } j < i-1 \\ 1 & \text{if } j = i-1 \end{cases} \end{aligned}$$

Easy to see that the causal graph of this problem forms a DAG, and an instance of this planning problem with the initial state $\langle 0, \ldots, 0\rangle$ and the goal state $\langle 0, \ldots, 0, 1\rangle$ have a unique minimal solution of length $2^n - 1$ corresponding to a Hamilton path in the state space.
■

The analysis of the proof of Lemma 4 was performed in order to point out the reason for this potential exponential escalation of the solution's size. An immediate conclusion of Lemma 4 is that there is a significant class domains with an acyclic causal graph for which planning is in NP.

Definition 1 *A causal graph is called* δ-path-restricted *if the number of different paths between every two nodes is bounded by* δ.

Lemma 5 *Plan generation for (STRIPS unary operator) planning problems with an underlying causal graph that is* δ*-path-restricted is in NP.*

Proof Based on the observations above

$$\textit{MinPlanSize}(\Pi) \leq \delta n^2$$

Again, we have found a class of planning problems that is in NP. But is it NP-hard? The following Lemma shows that in most cases, this is indeed the case.

Lemma 6 *Plan generation for propositional, 4-path-restricted planning problems is* NP*-complete.*

Proof The proof is by polynomial reduction from 3-SAT to the corresponding propositional, 4-path-restricted plan generation problems. 3-SAT is the problem of finding a satisfying assignment to a propositional formula in conjunctive normal form in which each conjunct (clause) has at most three literals.
Let $\mathcal{F} = C_1 \wedge \ldots \wedge C_n$ be a propositional formula belonging to 3-SAT, and let and $X_1, \ldots, X_m$ be the variables used in $\mathcal{F}$. An equivalent propositional, 4-path-restricted planning problem can be constructed as follows:

- $\mathcal{V} = \{A, B, X_1, \ldots, X_m, C_1, \ldots, C_n\}$
- *Parents*(A) = *Parents*$(B) = \{\emptyset\}$
- *Parents*$(X_1) = \ldots =$ *Parents*$(X_m) = \{A, B\}$
- *Parents*$(C_i) = \{A, B, X_{i_1}, X_{i_2}, X_{i_3}\}$, where X_{i_1}, X_{i_2}, and X_{i_3} are the variables that participate in the ith clause of $\mathcal{F}$.
- $Init$ - consist of false assignments to all variables in $\mathcal{V}$ ($\bar{v}$ for each $V \in \mathcal{V}$).
- $Goal$ - consist of true assignments to all variables in $\mathcal{V}$ (v for each $V \in \mathcal{V}$).

Let every operator $A \in \Lambda$ be presented as a three-tuple $\langle \{pre\}, \{post\}, \{prv\} \rangle$ of pre-, post-, and prevail conditions respectively. Then, the corresponding operator set Λ is specified as follows:

$$
\begin{array}{lll}
\Lambda_A & = \{ & \langle \{\bar{a}\}, \{a\}, \{\} \rangle \quad \} \\
\Lambda_B & = \{ & \langle \{\bar{b}\}, \{b\}, \{\} \rangle \quad \} \\
\vdots & & \\
\Lambda_{X_i} & = \{ & \langle \{\bar{x_i}\}, \{x_i\}, \{\bar{a}, \bar{b}\} \rangle, \\
 & & \langle \{\bar{x_i}\}, \{x_i\}, \{a, b\} \rangle \quad \} \\
\vdots & & \\
\Lambda_{C_i} & = \{ & \langle \{\bar{c_i}\}, \{c_i\}, \{\bar{a}, b, \alpha_1^i\} \rangle, \\
 & & \ldots \\
 & & \langle \{\bar{c_i}\}, \{c_i\}, \{\bar{a}, b, \alpha_{k_i}^i\} \rangle, \\
 & & \langle \{\bar{c_i}\}, \{c_i\}, \{a, \bar{b}, \alpha_1^i\} \rangle, \\
 & & \ldots \\
 & & \langle \{\bar{c_i}\}, \{c_i\}, \{a, \bar{b}, \alpha_{k_i}^i\} \rangle \quad \} \\
\vdots & &
\end{array}
$$

where $\alpha_1^i, \ldots \alpha_{k_i}^i$ are all possible truth assignments on the variables $X_{i_1}, X_{i_2}, X_{i_3}$, that satisfy the ith clause of $\mathcal{F}$. Easy to see, that the described planning problem have all propositional variables, single-effect operators and an acyclic causal graph (see figure 3).

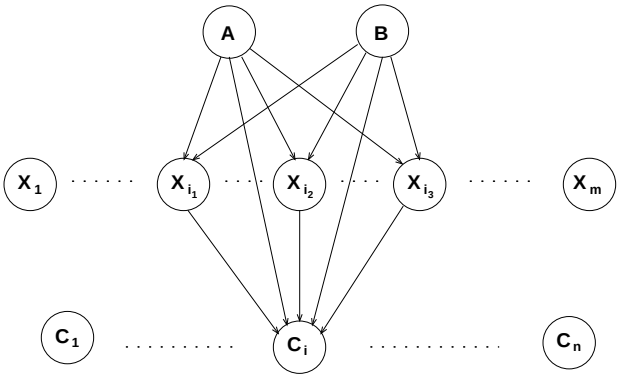


Figure 3. Causal graph of 3SAT satisfaction planning problem

First, we note that the resulting graph has at most 4 directed paths between any pair of nodes. Clearly, there are no paths between nodes A and B; exactly one path between A and B and any of the X_i's; at most one path between a particular X_i and a particular C_j; and exactly 4 paths between A or B and any of the C_i (one direct link, and three passing through the 3 variables consitituting C_i).
Now we describe the dynamics of the problem. As long as $A = \bar{a}$, and $B = \bar{b}$, each X_i can change its value from $\bar{x_i}$ to x_i, and no C_j can change its value. After either the value of A is changed to a, or the value of B is changed to b, no X_i can change its value (the assignment on the formula's variables is locked), but some variables from $C_1, \ldots, C_n$ can change their values from $\bar{c_i}$ to c_i. Finally, after the remaining variable from $\{A, B\}$ changes to its positive value, (i.e. the assignment on $\{A, B\}$ become a, b), each X_i, that still has the value $\bar{x_i}$ can be changed to x_i, but no C_j can change its value (the truth values of the formula's clauses depends only on previously locked values of the formula's variables).
Clearly, $Goal$ is reachable (Π is solvable) if and only if a satisfying assignment for $\mathcal{F}$ can be found. Likewise, the maximal number of paths between pairs of vertices in the causal graph is achieved between each locking variable (A and B), and each clause variable ($C_1, \ldots, C_n$), and is equal to 4. Thus, plan generation for propositional, 4-path-restricted planning problems is NP-hard, and from Lemma 5, we know that it is NP-complete.
■

3 SUMMARY

We have shown that the structure of the causal graph for unary operator STRIPS domains is an important factor in determining the computational complexity of plan generation. In particular, we have shown that a polynomial time algorithm exists for graphs in which there is at most one undirected path between nodes, and that in polytrees the maximal plan length is a low order polynomial. More generally, we have shown a relation between the number of path between variables in the causal graph and the computational complexity of the corresponding planning problem.

ACKNOWLEDGEMENTS

We would like to thank Samir Genaim for his assistance in one of the proofs, and the anonymous referee for useful remarks.

REFERENCES

[1] Christer Bäckström and Inger Klein, 'Planning in polynomial time: The SAS-PUBS class', *Computational Intelligence*, **7**(3), 181–197, (Aug 1991).
[2] Christer Bäckström and Bernhard Nebel, 'Complexity results for SAS^+ planning', *Computational Intelligence*, **11**(4), 625–655, (1995).
[3] B.C.Williams and P.P.Nayak, 'A reactive planner for model-based execution', in *Proceedings of the Fifteen International Joint Conference on Artificial Intelligence (IJCAI–97)*, (August 1997).
[4] C. Boutilier, R. Brafman, H. Hoos, and D. Poole, 'Reasoning with Conditional Ceteris Paribus Preference Statements', in *Proceedings of the Fifteenth Annual Conference on Uncertainty in Artificial Intelligence (UAI–99)*, (1999).
[5] Tom Bylander, 'The computational complexity of propositional STRIPS planning', *Artificial Intelligence*, **69**(1-2), 165–204, (1994).
[6] Daniel S. Weld, 'An introduction to least commitment planning', *AI Magazine*, (Summer 1994).

Heuristic Search Planning with BDDs

Stefan Edelkamp
Institut für Informatik
Am Flughafen 17
D-79110 Freiburg
edelkamp@informatik.uni-freiburg.de

Abstract. In this paper we study traditional and enhanced BDD-based exploration procedures capable of handling large planning problems. On the one hand, reachability analysis and model checking have eventually approached AI-Planning. Unfortunately, they typically rely on uninformed *blind* search. On the other hand, heuristic search and especially lower bound techniques have matured in effectively directing the exploration even for large problem spaces. Therefore, with heuristic symbolic search we address the unexplored middle ground between single state and symbolic planning engines to establish algorithms that can gain from both sides. To this end we implement and evaluate heuristics found in state-of-the-art heuristic single-state search planners.

1 Introduction

One currently very successful trend in deterministic fully-automated planning is heuristic single-state space search. The search space incorporates states as lists of instantiated predicates (also called atoms or fluents). The success of the heuristic search correlates with the quality of the estimate; the more informed the heuristic the better the achieved results. Heuristic search planners have outperformed other approaches on a sizable collection of deterministic domains. In the fully automated track of the AIPS-2000 planning competition[1] chaired by Fahim Baccus the System FF (by Hoffmann) was awarded for outstanding performance while HSP2 (by Geffner and Bonet), STAN (by Fox and Long), and MIPS (by Edelkamp and Helmert) were placed shared second.

The only available information on the implementation of remaining awarded planner *System R* (by Lin) is the contributed description by the author, reading as follows: *System R* is based on regression, and solves a goal one at a time. Briefly, given a conjunctive goal G, it chooses the first subgoal g that has not been satisfied yet in the current state, and work on it. Once it is achieved, say by P, it progresses the current state through P to a new current state, moves g to the end of G, and recursively tries to find a plan for the new G. When working on g, it regresses g over an action to a conjunctive goal G', and tries to achieve G' recursively. Subsequently, the solution quality in *System R* is not as good as for the other planners and has difficulties with some domains, where the goals were not serializable, but in some domains (like Logistics and Block's World) the system can cope with very large problem instances.

Historically, the first heuristic search planner was Bonet, Loerincs and Geffner's HSP [4], which also competed in AIPS-1998. HSP computes the heuristic values of a state by summing (or maximizing) depth values for each fluent for an overestimating (or admissible) estimate. These values are retrieved from the fix point of a relaxed exploration. Since the technique is similar to the first phase of building the layered graph structure in *Graphplan* (developed by Blum and Furst [2]), HSPr [6] (the suffix indicates a regression/backward search engine) has been extended to exclude so called *mutuals* similar to the original planning graph algorithm. In opposite to the parallel solutions obtained in *Graphplan*, HSP(r) produce sequential solutions. In the most recent extension to the planner, Haslum and Geffner [27] generalize the (admissible) estimator by dynamic programming parameterized with an order value m. For large values of m the estimate h^m converges to the optimal heuristic estimate h^*. In case $m = 1$ the new estimator reduces to maximizing the fluent values, for $m = 2$ the authors introduce the *max-pair heuristic* computing a distance value to the goal for each pair of atoms. This is the heuristic incorporated in Geffner and Bonet's planner HSP2 at competition time. The underlying search algorithm is a weighted version of IDA* [42], scaling the heuristic with respect to the generated path length with a factor of two for a better performance by the cost of non-optimal solutions. Due to the observed overhead at run-time, high-order heuristics have not been applied yet. As a straight forward extension to the *max-pair heuristic* it might be conjectured that for pairs of fluents and their corresponding relaxed solution length a weighted bipartite minimum-matching algorithm (available in cubic time) as applied in Sokoban [31] might lead to a better lower bound approximation.

The success of HSP has inspired the planners GRT by Refanidis and Vlahavas [43] and FF by Hoffmann [28] and influenced the development of the planners STAN and MIPS.

GRT abbreviates a heuristic search planner based on *greedy regression tables* to trace the responsibility of a fact being achieved. The inference of a heuristic value for each state is thus found in a backward analysis of the fluent space. The *regression table* is closely related to *regression-match graphs* [40] estimating the goal distance of a state. This approach ignores any conflict and then counts the minimal number of steps. Despite new ideas such as *Exploiting State Constraints* [27], in AIPS-2000 the heuristic of GRT was too weak to compete with the improvements applied in HSP2 and in FF.

The winner FF (for fast-forward planning) solves a relaxed planning problem for *every* encountered state in a combined forward *and* backward traversal. Therefore, the *FF-Heuristic* is an elaboration to the *HSP-Heuristic*, since the latter only considers the first phase. The efforts in computing a very accurate heuristic estimate correlates with data in solving single agent challenges like the 24-Puzzle [37],

[1] See `http://www.cs.toronto.edu/aips2000` for details.

Sokoban [31], and Rubik's Cube [36] and suggests that even involved work for improving the heuristic pays off. With *enforced hill climbing* it further employs another search strategy and drastically reduces the explored portion of search space. It makes use of the fact that phenomena like big plateaus or local minima — with respect to the heuristic described above — do not occur very often in benchmark planning problems.

STAN's success is due to building a hybrid of two strategies: The original GRAPHPLAN-based STAN algorithm and a forward planner using a heuristic function based on the length of the relaxed plan (as in HSP and FF). STAN makes is the use of domain analysis techniques to select automatically between these strategies. Therefore, the major contribution is the automatic synthesis and use of *generic types* to choose an appropriate algorithm for the specified problem instance at hand [39].

An orthogonal approach in tackling huge search spaces is a symbolic representation of sets of states. The SATPLAN approach by Kautz and Selman [32] has shown that representional issues can be resolved by parsing the planning domain into a collection of Boolean formulae (one for each depth level). The system BLACKBOX (a hybrid planner based on merging SATPLAN with GRAPHPLAN [33]) performed well on AIPS-1998, but failed to solve as many problems as the heuristic search planners on the domains in AIPS-2000. However, it should be denoted that the results of SATPLAN (GRAPHPLAN) are optimal in the number of sequential (parallel) steps, while heuristic search planners tend to overestimate in order to cope with state space sizes of 10^{20} and beyond.

Although efficient satisfiability solvers have been developed in the last decade, the blow-up in the size of the formulae even for simple planning domains calls for a concise representation. This leads to reduced ordered binary decision diagrams (BDDs) [7], an efficient data structure for Boolean functions. Through their unique representation BDDs are effectively applied to the synthesis and verification of hardware circuits [8] and incorporated within the area of *model checking* [9]. Nowadays BDDs are a fundamental tool in various research areas of computer science and very recently BDDs are encountering AI-research topics like *heuristic search* [21] and *planning* [25]. The diverse research aspects of *traditional STRIPS planning* [22], *non-deterministic planning* [10], *universal planning* [12], and *conformant planning* [11] indicate the wide range of BDD-related planning.

Our planner MIPS uses BDDs to compactly store and maintain sets of propositionally represented states. The concise state representation is inferred in an analysis prior to the search and, by utilizing this representation, accurate reachability analysis and backward chaining are carried out without necessarily encountering exponential representation explosion. MIPS was originally designed to prove that BDD-based exploration methods are an efficient means for implementing a domain-independent planning system with some nice features, especially guaranteed optimality of the plans generated. If problems become harder and information on the solution length is available, MIPS invokes its incorporated heuristic single state search engine (similar to FF), thus featuring two entirely different planning algorithms, aimed to assist each other on the same state representation. Note that implementation issues of MIPS are discussed in [20].

The other two BDD planners in AIPS-2000, BDDPLAN by Stör [29] and *PROPPLAN* by Fourman [24], lack the precompiling phase of MIPS, therefore, were too slow for traditional STRIPS problems. Moreover a single state extension to their planners is not being provided. In the generalized ADL settings PROPPLAN has proven to be competive even with the FF approach, which solves more problems in less time, but fails to find optimal solutions.

This paper extends the idea of BDD representations and exploration in the context of heuristic search. The heuristic estimate is based on subpositions (called patterns) calculated prior to the search representing all (state,estimate)-pairs in one BDD. Therefore, the heuristic is a form of a pattern database with planning patterns corresponding to (one or a collection of) fluents. This heuristic will be integrated into a previously published BDD-based version of the A* algorithm [26], called BDDA* [21]. Moreover, we alter the concept of BDDA* to *pure heuristic search* which seems to be more suited at least to some planning problems. Thereby, we allow non-optimistic heuristics and sacrifice optimality but succeed in searching larger problem spaces.

We have structured the paper as follows: First of all, we give a simple planning example and briefly introduce BDDs basics. Thereafter, we turn to the exploration algorithms, starting with blind search then turning to the directed approach BDDA*, its adaption to planning, and its refinement for *pure heuristic search*. We end with some experimental data and draw conclusions.

2 BDD Representation

Let us consider a simple example of a planning problem for a truck to deliver one package from Los Angeles to San Francisco. The initial state (in STRIPS like notation) is given by `(PACKAGE package)`, `(TRUCK truck)`, `(LOCATION los-angeles)`, `(LOCATION san-francisco)`, `(AT package los-angeles)`, and `(AT truck los-angeles)` while the goal state is specified by `(AT package san-francisco)`. We have three operator schemas in the domain, namely `LOAD` (for loading a truck with a certain package at a certain location), `UNLOAD` (the inverse operation), and `DRIVE` (a certain truck from one city to another). The operator schemas are expressed in form of preconditions and effects.

The precompiler to infer a small state encoding consists of three phases [19]. In a first *constant predicate* phase it observes that the predicates `PACKAGE`, `TRUCK` and `LOCATION` remain unchanged by the operators. In the next *merging* phase the precompiler determines that `at` and `in` should be encoded together, since a `PACKAGE` can exclusively be at a `LOCATION` or in a `TRUCK`. By *fact space exploration* (a simplified but complete exploration of the planning space) the following fluent facts are generated: `(AT package los-angeles)`, `(AT package san-francisco)`, `(AT truck los-angeles)`, `(AT truck san-francisco)`, and `(IN package truck)`. This leads to a total encoding length of three bits. Using two bits x_0 and x_1 the fluents `(AT package los-angeles)`, `(AT package san-francisco)`, and `(IN package truck)` are encoded with 00, 01, and 10, respectively, while the variable x_2 represents the fluents `(AT truck los-angeles)` and `(AT truck san-francisco)`.

Therefore, a Boolean representation of the start state is given by $\overline{x_0} \wedge \overline{x_1} \wedge \overline{x_2}$ while the set of goal states is simply formalized with the expression $\overline{x_0} \wedge x_1$. More generally, for a set of states S the *characteristic function* $\phi_S(a)$ evaluates to *true* if a is the binary encoding of one state x in S. As the formulae for the start and the goal states indicate, the symbolic representation for a large set of states is typically smaller than the cardinality of the represented set.

Since the satisfiability problem for Boolean formulae is NP hard, binary decision diagrams are used to for their efficient and unique graph representation. The nodes in the directed acyclic graph structure are labeled with the variables to be tested. Two outgoing edges labeled *true* and *false* direct the evaluation process with the result

found at one of the two sinks. We assume a fixed variable ordering on every path from the root node to the sink and that each variable is tested at most once. The BDD size can be exponential in the number of variables but, fortunately, this effect rarely appears in practice. The satisfiability test is trivial and given two BDDs G_f and G_g and a Boolean operator $\otimes$, the BDD $G_{f \otimes g}$ can be computed efficiently. The most important operation for exploration is the *relational product* of a vector of variables v and two Boolean functions f and g. It is defined as $\exists v\ (f \wedge g)$. Since existential quantification of one variable x_i in a Boolean function f is equal to disjunction $f_{\overline{x_i}} \vee f_{x_i}$, the quantification of v results in a sequence of subproblem disjunctions. Although computing the relational product is NP-hard in general, specialized algorithms have been developed leading to an efficient computation for many practical applications.

An operator can also be seen as an encoding of a set. The *transition relation* T is defined as the disjunction of the characteristic functions of all pairs (x', x) with x' being the predecessor of x. For the example problem, `(LOAD package truck los-angeles)` corresponds to the pair $(00|0, 10|0)$ and `(LOAD package truck san-francisco)` to $(01|1, 10|1)$. Subsequently, the `UNLOAD` operator is given by $(10|0, 00|0)$ and $(10|1, 10|1)$. The `DRIVE` action for the truck is represented by the strings $(00|*, 00|*)$ $(01|*, 01|*)$, and $(10*, 10|*)$ with $* \in \{0, 1\}$. For a concise BDD representation of the transition relation (cf. Figure 1) the variable ordering is chosen that the set of variable in x' and x are *interleaved*, i.e. given in alternating order.

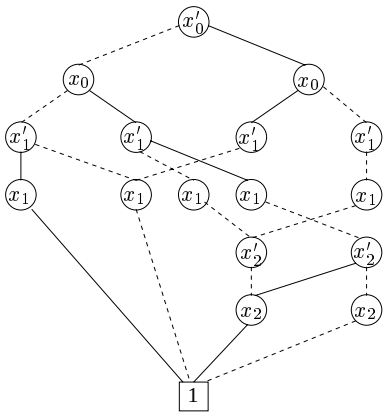


Figure 1. The transition relation for the example problem. For the sake of clarity, the *false* sink has been omitted. Dashed lines and solid lines indicate edges labeled *false* and *true*, respectively.

The *weighted transition relation* $T(w, x', x)$ is a straight-forward extension to weighted problem graphs and evaluates to 1 if and only if the step from x' to x has costs w (encoded in binary).

3 BDD-Based Blind Search

Let S_i be the set of states reachable from the initial state s in i steps, initialized by $S_0 = \{s\}$. The following equation determines ϕ_{S_i} given both $\phi_{S_{i-1}}$ and the transition relation:

$$\phi_{S_i}(x) = \exists x'\ (\phi_{S_{i-1}}(x') \wedge T(x', x)).$$

The formula calculating the successor function is a relational product. A state x belongs to S_i if it has a predecessor x' in the set S_{i-1} and there exists an operator which transforms x' into x. Note that on the right hand side of the equation ϕ depends on x' compared to x on the left hand side. Thus, it is necessary to substitute x with x' in ϕ_{S_i} beforehand, which can be achieved by a simple textual replacement of the node labels in the diagram structure.

In order to terminate the search, we successively test, whether a state is represented in the intersection of the set S_i and the set of goal states G by testing the identity of $\phi_{S_i} \wedge \phi_G$ with the trivial zero function. Since we enumerated $S_0, \ldots, S_{i-1}$ the iteration index i is known to be the optimal solution length.

Let *Open* be the representation of the search horizon and *Succ* the BDD for the set of successors. Then the algorithm can be realized as the pseudo-code Figure 2 suggests.

```
procedure Breadth-First-Search
  Open ← φ_{s}
  do
    Succ ← ∃x' (Open(x') ∧ T(x', x))
    Open ← Succ
  while (Open ∧ φ_G ≡ 0)
```

Figure 2. Breadth-first search implemented with BDDs.

This simulates a breadth-first exploration and leads to three iterations for the example problem. We start with the initial state represented by a BDD of three inner nodes for the function $\overline{x_0} \wedge \overline{x_1} \wedge \overline{x_2}$. After the first iteration we get a BDD size of four representing three states and the function $(\overline{x_0} \wedge \overline{x_1}) \vee (x_0 \wedge \overline{x_1} \wedge \overline{x_2})$. The next iteration leads to four states in a BDD of one internal node for $\overline{x_1}$, while the last iteration results in a BDD containing a goal state.

3.1 Bidirectional Search

We start with the goal set $B_0 = G$ and iterate until we encounter the start state. In backward search we take advantage of the fact that T has been defined as a relation. Therefore, we iterate according to the formula $\phi_{B_i}(x') = \exists x\ (\phi_{B_{i-1}}(x) \wedge T(x', x))$. In bidirectional breadth-first search forward and backward search are carried out concurrently. On the one hand we have the forward search frontier F_f with $F_0 = \{s\}$ and on the other hand the backward search frontier B_b with $B_0 = G$. When the two search frontiers meet $(\phi_{F_f} \wedge \phi_{B_b} \not\equiv 0)$ we have found an optimal solution of length $f + b$. With the two horizons *fOpen* and *bOpen* the algorithm can be implemented as shown in Figure 3.

The choice of the search direction (function call `forward`) is crucial for a successful exploration. There are three simple criteria: BDD size, the number of represented states, and smaller exploration time. Since the former two are not well suitable to predict the computational efforts of the next iteration the third criterion is preferred.

3.2 Forward Set Simplification

The introduction of a list *Closed* containing all states ever expanded is a very common approach in single state exploration to avoid duplicates in the search. Usually, the memory structure is realized as a hash table, which in this context is referred by the term *transposition table*. For symbolic search this technique is called *forward set simplification* (cf. Figure 4).

procedure *Bidirectional Breadth-First-Search*
  fOpen $\leftarrow \phi_{\{s\}}$; *bOpen* $\leftarrow \phi_G$
  do
    if (*forward*())
      Succ $\leftarrow \exists x' \ (fOpen(x') \wedge T(x', x))$
      fOpen $\leftarrow$ *Succ*
    else
      Succ $\leftarrow \exists x \ (bOpen(x) \wedge T(x', x))$
      bOpen $\leftarrow$ *Succ*
  while (*fOpen* $\wedge$ *bOpen* $\equiv 0$)

Figure 3. Bidirectional Breadth-first search implemented with BDDs.

procedure *Forward Set Simplification*
  Closed $\leftarrow$ *Open* $\leftarrow \phi_{\{s\}}$
  do
    Succ $\leftarrow \exists x' \ (Open(x') \wedge T(x', x))$
    Open $\leftarrow$ *Succ* $\wedge \neg$ *Closed*
    Closed $\leftarrow$ *Closed* $\vee$ *Succ*
  while (*Open* $\wedge \phi_G \equiv 0$)

Figure 4. Breadth-first search with *forward set simplification* implemented with BDDs.

The effect in the given example is that after the first iteration the number of states shrinks from three to two while the new BDD for $(\overline{x_0} \wedge \overline{x_1} \wedge x_2) \vee (x_0 \wedge \overline{x_1} \wedge \overline{x_2})$ has five inner nodes. For the second iteration only one newly encountered state is left with three inner BDD nodes representing $x_0 \wedge \overline{x_1} \wedge \overline{x_2}$. Forward set simplification is also used to terminate the search in case of failure in a complete planning space exploration. Note that any set in between the successor set *Succ* and the simplified successor set *Succ* $-$ *Closed* will be a valid choice for the horizon *Open* in the next iteration. Therefore, one may choose a set R that minimizes the BDD representation instead of minimizing the set of represented states. Without going into involved details we denote that such image size optimizing operators are available in several BDD packages [14].

4 BDD-Based Directed Search

Before turning to the BDD-based algorithm for directed search we take a brief look at Dijkstra's single-source shortest path algorithm, *Dijkstra* for short, which finds a solution path with minimal length within a weighted problem graph [17]. *Dijkstra* differs from breadth-first search in ranking the states next to be expanded. A priority queue is used, in which the states are ordered with respect to an increasing f-value. Initially, the queue contains only the initial state s. In each step the state with the minimum merit f is dequeued and expanded. Then the successor states are inserted into the queue according to their newly determined f-value. The algorithm terminates when the dequeued element is a goal state. The f-value of this state is the length of the minimal solution path.

As said, BDDs allow sets of states to be represented very efficiently. Therefore, the priority queue *Open* can be represented by a BDD based on tuples of the form (*value*, *state*). The variables should be ordered in a way which allows the most significant variables to be tested at the top. The variables for the encoding of the *value* should have smaller indices than the variables encoding the *state*, since this encoding leads to small BDDs and allows an intuitive understanding of the BDD and its association with the priority queue.

procedure Symbolic-Version-of-Dijkstra
  Open$(f, x) \leftarrow \ (f = 0) \wedge \phi_{S^0}(x)$
  do
    $f_{\min} = \min\{f \mid f \wedge Open \neq \emptyset\}$
    Min$(x) \leftarrow \exists f \ ($*Open* $\wedge \ f = f_{\min})$
    Rest$(f, x) \leftarrow$ *Open* $\wedge \neg$ *Min*
    Succ$(f, x) \leftarrow \exists x', w \ ($*Min*$(x') \wedge$
        $T(w, x', x) \wedge add(f_{\min}, w, f))$
    Open $\leftarrow$ *Rest* $\vee$ *Succ*
  while (*Open* $\wedge \phi_G \equiv 0$)

Figure 5. Dijkstra's single-source shortest-path algorithm implemented with BDDs.

The symbolic version of Dijkstra (cf. Figure 5) now reads as follows. The BDD *Open* is set to the representation of the start state with value zero. Until we find a goal state in each iteration we extract *all* states with minimal f-value $f_{\min}$, determine the successor set and update the priority queue. Successively, we compute the minimal f-value $f_{\min}$, the BDD *Min* of all states in the priority queue with value $f_{\min}$, and the BDD of the remaining set of states. If no goal state is found, the variables in *Min* are substituted as above before the (weighted) transition relation $T(w, x', x)$ is applied to determine the BDD for the set of successor states. To attach new f-values to this set we have to retain the old f-value $f_{\min}$ and to calculate $f = f_{\min} + w$. Finally, the BDD *Open* for the next iteration is obtained by the disjunction of the successor set with the remaining queue.

It remains to show how to perform the arithmetics using BDDs. Since the f-values are restricted to a finite domain, the Boolean function *add* with parameters a, b and c can be built being *true* if c is equal to the sum of a and b. A recursive calculation of *add*(a, b, c) should be prefered:

$$add(a, b, c) = ((b = 0) \wedge (a = c)) \vee$$
$$\exists \ b', c' \ (inc(b', b) \ \wedge \ inc(c', c) \ \wedge \ add(a, b', c')),$$

with *inc* representing all pairs of the form $(i, i + 1)$. Therefore, symbolic breadth-first-search (with forward set simplification) can be applied to determine the fixpoint of *add* (subject to a certain predefined finite domain of the variables).

4.1 Heuristic Pattern Databases

For symbolically constructing the heuristic function a simplification T' to the transition relation T that regains tractability of the state space is desirable. However, obvious simplification rules might not be available. Therefore, in heuristic search we often consider relaxations of the problem that result in subpositions. More formally, a state v is *subposition* of another state u if and only if the characteristic function of u logically implies the characteristic function of v, e.g., $\phi_{\{u\}} = \overline{x_1} \wedge x_2 \wedge x_3 \wedge \overline{x_4} \wedge x_5$ and $\phi_{\{v\}} = x_2 \wedge x_3$ results in $\phi_{\{u\}} \Rightarrow \phi_{\{v\}}$. As a simple example take the Manhattan distance

in sliding tile solitaire games like the famous Fifteen-puzzle. It is the sum of solutions of single tile problems that occur in the overall puzzle. The improvement of the Manhatten distance incorporates linar conflicts due to the interplay of two tiles has lead to solutions to random instances of the 24-Puzzle with a state space of $25!/2 \approx 10^{25}$ states.

More generally, a *heuristic pattern data base* is a collection of pairs of the form (*estimate, pattern*) found by optimally solving problem relaxations that respect the subposition property [15]. The solution lengths of the patterns are then combined to an overall heuristic by taking the maximum (usually leading to an admissible heuristic) or the sum of the individual values (in which case we get an overestimization).

Heuristic pattern data bases have been effectively applied in the domains of Sokoban [31], to the Fifteen-Puzzle [15], and to Rubik's Cube [36]. In single-state search heuristic pattern databases are implemented by hash table, but in symbolic search we have to construct the estimator symollically, only using logical combinators and Boolean quantification.

Since heuristic search itself can be considered as the matter of introducing lower bound relaxations into the search process, in the following we will maximize the relaxed solution path values. The maximizing relation *max*(a, b, c), evaluates to 1 if c is the maximum of a and b and is based on the relation *greater*, since

$$\textit{max}(a, b, c) = (\textit{greater}(a, b) \wedge (a = c)) \vee$$
$$(\neg \textit{greater}(a, b) \wedge (b = c))$$

.

The relation *greater*(a, b) itself might be implemeted by existential quantifying the add relation:

$$\textit{greater}(a, b) = \exists t\ \textit{add}(b, t, a)$$

Next we will find a way to automatically infer the heuristic estimate. To combine n fluent pattern $p_1, \ldots, p_n$ with estimated distances $d_1, \ldots, d_n$ to the goal we use $n+1$ additional slack variables $t_0, \ldots, t_n$ which are existenially quantified later on. We define subfunctions H_i of the form

$$H_i(t_i, t_{i+1}, \textit{state}) = (\neg\, p_i \wedge (t_i = t_{i+1})) \vee$$
$$(p_i \wedge \textit{max}(d_i, t_i, t_{i+1})),$$

with $H_i(t_i, t_{i+1}, \textit{state})$ denoting the following relation: If the accumulated heuristic value up to fluent i is t_i, then the accumulated value including fluent i is t_{i+1}. Therefore, we can combine the subfunctions to the overall heuristic estimate as follows:

$$H(\textit{estimate,state}) = \exists\, t_1, \ldots, t_n\ (t_0 = 0) \wedge H(t_n, \textit{estimate,state}) \wedge$$
$$\bigwedge_{i=0}^{n-1} H_i(t_i, t_{i+1}, \textit{state})$$

.

In some problem graphs subpositions or patterns might constitute a feature in which every position containing it is unsolvable. These *deadlocks* are frequent in directed search problems like Sokoban and can be learned domain or problem specifically. Deadlocks are heuristic patterns with a infinite heuristic estimate. Therefore, a deadlock table DT is the disjunction of the characteristic functions according to subpositions that are unsolvable.

The integration of *deadlock tables* in the search algorithm is quite simple. For the BDD for DT we assign the new horizon *Open* as

$$\textit{Open} \wedge \neg(\textit{Open} \Rightarrow \textit{DT})$$

which is equivalent to

$$\textit{Open} \leftarrow \textit{Open} \wedge \neg \textit{DT}$$

.

4.2 Two Different Heuristics

Patterns in planning are fluents. The estimated distance of each single fluent p to the goal is a heuristic value associated with p. We examine two heuristics.

4.2.1 HSP-Heuristic:

In HSP the values are recursively calculated by the formula $h(p) = min\{h(p), 1 + h(C)\}$ where $h(C)$ is the cost of achieving the conjunct C, which in case of HSPr is the list of preconditions. For determining the heuristic the planning space has been simplified by omitting the delete effects. The algorithms in HSP and HSPr are variants of pure heuristic search incorporated with restarts, plateau moves, and overestimation.

The exploration phase to minimize the state description length in our planner has been extended to output an estimate $h(p)$ for each fluent p. Since we avoid duplicate fluents in the breadth-first *fact-space-exploration*, with each encountered fluent we associate a depth by adding the value 1 to its predecessor. The quality of the achieved distance values are not as good as in HSPr since we are not concerned about mutual exclusions in any form. Giving the list of value/fluent pairs a symbolic representation of the sub-relations and the overall heuristic is computed.

In the example we compute that `(AT ball los-angeles)` and `(AT truck los-angeles)` have a distance of zero from the initial state `(AT truck san-francisco) (IN ball truck)` have a depth of one and `(AT ball san-francisco)` has depth two. Figure 6 depicts the BDD representation of the overall heuristic function for the example.

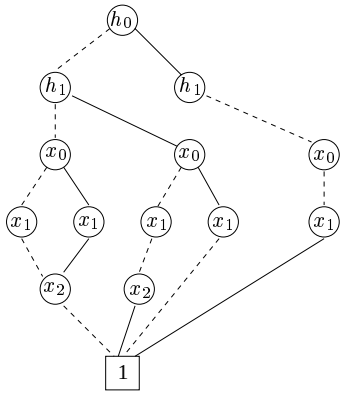


Figure 6. The BDD representation for the heuristic function in the example problem. In this case the individual pattern values have been maximized.

4.2.2 FF-Heuristic:

The main idea of FF is fairly simple: Solve the relaxed planning problem (delete-facts omitted) with GRAPHPLAN on-line for each state, i.e., build the plan graph *and* extract a simplified solution by counting the number of instantiated operators that at least have to fire. This is the heuristic value. By relaxed forward and backward search one state can usually be evaluated in less than ten milliseconds.

Since the branching factor is large (one state has up to hundreds of successors) by determining *helpful actions*, only a *relevant* part of all successors is considered. The overall search phase is entitled *enforced hill-climbing*. Until the next smaller heuristic value is found a breadth first search is invoked. Then the search process iterates with one state evaluating to this value.

In our planner we have (re-)implemented the FF-approach both to have an efficient heuristic single-state search engine at hand and to build an improved estimate for symbolic search. Since the FF approach is based on states and not on fluents, we cannot directly infer a symbolic version of the heuristic. We have to weaken the state-dependent character of the heuristic down to fluents. Moreover, simplifying the start state to a fluent may give no heuristic value at all, since the goal will not necessarily be reached by the relaxed exploration. Therefore, the estimate for each fluent is calculated by partitioning the goal state instead. Since we get improved distance estimates with respect to the initial state, we obtain a heuristic for backward search. However this is no limitation, since the concept of STRIPS operators can be inverted, yielding a heuristic in the usual direction.

In case of *Block's World*, any heuristic based on fluent values is misleading, since if the block at the bottom is not correctly placed even states with all but one satisfied subgoals are far off from the goal state. To cope with hat problem Hoffmann proposes two different goal ordering strategies, both based knowledge-gathering based on exploration [35].

4.3 BDDA*

In *informed search* with every state in the search space we associate a lower bound estimate h. By reweighting the edges the algorithm of Dijkstra can be transformed into A*. The new weight $\hat{w}$ is set to the old one w minus the h-value of the source node x', plus the value of the target node x resulting in the equation $\hat{w}(x', x) = w(x', x) - h(x') + h(x)$. The length of the shortest paths will be preserved and no new negative weighted cycle is introduced [13]. More formally, if we denote $\delta(s, g)$ for the length of the shortest path from s to a goal state g in the original graph, and $\hat{\delta}(s, g)$ the shortest path in the reweigthed graph then $w(p) = \delta(s, g)$ if and only if $\hat{w}(p) = \hat{\delta}(s, g)$.

The rank of a node is the combined value $f = g + h$ of the generating path length g and the estimate h. The information h allows us to search in the direction of the goal and its quality mainly influences the number of nodes to be expanded until the goal is reached.

In the symbolic version of A*, called BDDA*, the relational product algorithm determines all successor states in one evaluation step. It remains to determine their values. For the dequeued state x' in A* we have $f(x') = g(x') + h(x')$. Since we can access f, but usually not g, the new value $f(x)$ of a successor x has to be calculated in the following way

$$f(x) = g(x) + h(x) = g(x') + w(x', x) + h(x) =$$
$$f(x') + w(x', x) - h(x') + h(x).$$

The estimator H can be seen as a relation of tuples (*estimate, state*) which is *true* if and only if h(*state*)=*estimate*. We assume that H can be represented as a BDD for the entire problem space. The cost values of the successor set are calculated according to the equation mentioned above. The arithmetics for *formula*(h', h, w, f', f) based on the old and new heuristic value (h' and h, respectively), and the old and new merit (f' and f, respectively) are given as follows.

$$\textit{formula}(h', h, w, f', f) = \exists\, t_1, t_2\ add(t_1, h', f') \wedge$$
$$add(t_1, w, t_2)\ \wedge\ add(h, t_2, f).$$

The implementation of the algorithm BDDA* is depicted in Figure 7.

procedure BDDA*
Open$(f, x) \leftarrow H(f, x) \wedge \phi_{S^0}(x)$
do
  $f_{\min} = \min\{f \mid f \wedge \textit{Open} \neq \emptyset\}$
  Min$(x) \leftarrow \exists f\ (\textit{Open} \wedge f = f_{\min})$
  Rest$(f, x) \leftarrow \textit{Open} \wedge \neg\, \textit{Min}$
  Succ$(f, x) \leftarrow \exists w, x'\ (\textit{Min}(x') \wedge T(w, x', x) \wedge$
    $\exists h'\ (H(h', x') \wedge \exists h\ (H(h, x) \wedge$
    formula$(h', h, w, f_{\min}, f))))$
  Open $\leftarrow$ *Rest* $\vee$ *Succ*
while (*Open* $\wedge\ \phi_G \equiv 0$)

Figure 7. Hart, Nilsson and Raphael's A* algorithm implemented with BDDs.

Since all successor states are reinserted in the queue we expand the search tree in best-first manner. Optimality and completeness is inherited by the fact that given an optimistic heuristic A* will find an optimal solution.

Given a uniform weighted problem graph and a consistent heuristic the worst-case number of iterations in BDDA* is $O(f^{*2})$, with f^* being the optimal solution length [21].

Preliminary results of BDDA* even in handcoded traditional single-agent search domains are promising.

In (a moderately difficult instance to) the Fifteen-Puzzle, the 4×4 version of the well-known sliding-tile $(n^2 - 1)$-Puzzles, a minimal solution of 45 moves was found by BDDA* within 176 iterations with a maximal BDD-size of 215.000 nodes representing 136.000 states. With a breadth-first search approach it was impossible to find any solutions because of memory limitations. Already after 19 iteration-steps more than 1 million BDD-nodes were needed to represent more than 1.4 million states. Note, that the upper limit of (domain independent) planners are to solve some instances to the Eight-Puzzle [41].

Sokoban was considered as a domain for AIPS-2000, but was in favor of *Freecell*, the Window solitaire card game. To find the minimal solution in Sokoban an efficient encoding is essential. There are 56 different fields available for the man, resulting in a binary encoding of six bits. For the balls 23 positions are either not reachable or the configuration becomes unsolvable. Therefore, 33 bits are sufficient to specify for each considerable position if a ball is placed on it or not. The BDDA* algorithm was invoked with a very poor heuristic, counting the number of balls not on a goal position.

Breadth first search finds the optimal solution with a peak BDD of 75,000 nodes representing 8,400,00 states in the optimal number of

230 iterations. BDDA* with the heuristic leads to 419 iterations and to a peak BDD of 68,000 nodes representing 4,300,00 states. Note that even with such a poor heuristic, the number of nodes expanded by BDDA* is significantly smaller than in a breadth-first-search approach and their representation is more memory efficient. The number of represented states is up to 250 times higher than the number of necessary BDD nodes. Additionally, more bits are needed for the encoding of a state than for the encoding of a BDD node.

4.4 Pure BDDA*

A variant of BDDA*, called *Pure* BDDA*, can be obtained by ordering the priority queue only according to the h values. In this case the calculation of the successor relation simplifies to $\exists x'\ (\mathit{Min}(x') \wedge T(x', x) \wedge H(f, x))$ as shown in Figure 8.

procedure Pure BDDA*
Open$\leftarrow H(f, x) \wedge \phi_{S^0}$
do
  $f_{\min} = \min\{f \mid f \wedge \mathit{Open} \neq \emptyset\}$
  $\mathit{Min}(x) \leftarrow \exists f\ \mathit{Open} \wedge f = f_{\min}$
  $\mathit{Rest}(f, x) \leftarrow \mathit{Open} \wedge \neg\ \mathit{Min}$
  $\mathit{Succ} \leftarrow \exists x'\ (\mathit{Min}(x') \wedge T(x', x) \wedge H(f, x))$
  $\mathit{Open} \leftarrow \mathit{Rest} \vee \mathit{Succ}$
while ($\mathit{Open} \wedge \phi_G \equiv 0$)

Figure 8. Pure BDDA* algorithm implemented with BDDs.

The old f-value will be overwritten and need not to be provided. Therefore, *Pure* BDDA* is a greedy hill climber.

Unfortunately, even for an optimistic heuristic the algorithm is not admissible and, therefore, will not necessarily find an optimal solution. The hope is that in huge problem spaces the estimate is good enough to lead the solver into a promising goal direction. Therefore, especially heuristics with overestimizations can support this aim.

On solution paths the heuristic values eventually decrease. Hence, in *Pure* BDDA* we take advantage of the fact that the most promising states are in the front of the priority queue, have a smaller BDD representation, and are explored first. This compares to BDDA* in which the combined merit on the solution paths eventually increases. The advantage of symbolic representation compared to single state exploration is that several paths are searched in parallel.

Note the similarity of considering a possibly large set of state in *enforced hill climbing* as implemented in FF. A good trade-off between exploitation and exploration has to be found. In FF breadth-first search for the next heuristic estimate consolidates pure heuristic search for a complete search strategy.

Figure 9 depicts the different dequeued BDDs *Min* together with their heuristic valuation in the exploration phase of *Pure* BDDA* for the example problem.

5 Experiments

From given results on the different heuristic search planners [28] it can be obtained that heuristics pay off best in the *Gripper* and the *Logistics* domain. We add some data obtained in two model checking planning problems.

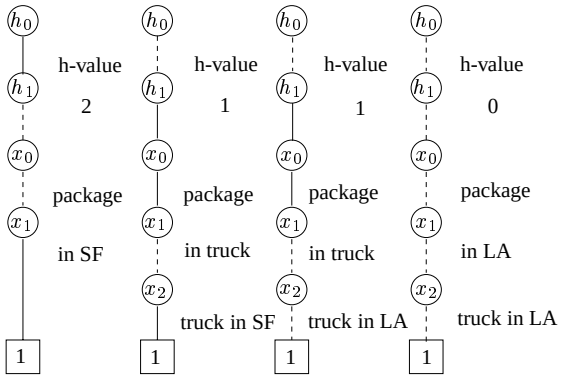


Figure 9. Backward exploration of the example problem in *Pure* BDDA*. In each iteration step the BDD *Min* with associated h-value is shown. Note that when using forward set simplification these BDDs additionally correspond to a snapshot of the priority queue *Open*.

More experimental results on AIPS-1998 results are provided in [20]. The performance on AIPS-2000 can be obtained at the official homepage of the competition.

5.1 Gripper

The effect of forward set simplification and optimization can best be studied in the scalable *Gripper* domain depicted in Table 1[2].

B abbreviates *bidirectional search*, O BDD image *optimization*, and F *forward set simplification*, respectively. When the problem instances get larger the additional computations pay off. In *Gripper* bidirectional search leads to no advantage since due to the symmetry of the problem the climax of the BDD sizes is achieved in the middle of the exploration. This is an important advantage to BDD-based exploration: Although the number of states grows continuously, the BDD representation might settle and become smaller. The data further suggests that optimizing the BDD structure with the proposed optimization is helpful only in large problems.

	Solution Length	BFS	+B	+BF	+BFO
1-1	11	0.00	0.01	0.01	0.01
1-2	17	0.01	0.01	0.02	0.02
1-3	23	0.02	0.03	0.02	0.02
1-4	29	0.03	0.03	0.04	0.04
1-5	35	0.04	0.04	0.07	0.07
1-6	41	0.06	0.06	0.08	0.08
1-7	47	0.08	0.08	0.11	0.14
1-8	53	0.12	0.13	0.19	0.20
1-9	59	0.35	0.36	1.33	1.58
1-10	65	0.72	1.93	2.06	2.15
1-11	71	1.27	2.33	2.36	2.43
1-12	77	1.95	3.21	3.05	3.13
1-13	83	2.80	3.91	3.48	3.49
1-14	89	3.80	5.04	4.28	4.36
1-15	95	4.93	6.26	5.29	5.43
1-16	101	6.32	7.21	6.41	6.07
1-17	107	7.72	8.94	7.26	7.52
1-18	113	9.82	10.91	8.65	8.61
1-19	119	24.73	26.11	15.28	15.35
1-20	125	34.59	36.73	20.41	20.08

Table 1. Searching the Gripper domain with breadth-first search, combined with bidirectional search, forward set simplification and optimization.

[2] The CPU-times in the experiments are given in seconds on a Linux-PC (Pentium III/450 MHz/128 MByte).

As we can see, *Gripper* is not a problem to *BDD*-based search at all, whereas it is hard for *Graphplan* search engines.

5.2 Logistics

Due to the first round results in AIPS-2000 it can be deduced that FF's, STAN's and MIPS's heuristic single search engine are state-of-the-art in this domain, but Logistics problems turn out to be suprisingly hard for BDD exploration and therefore a good benchmark domain for BDD inventions. For example Jensen's BDD-based planning system, called UMOP, fails to solve any of the AIPS-1998 (first-round) problems [30] and breadth-first search in *MIPS* yields only two domains to be solved optimally.

This is due to high parallelism in the plans, since optimal parallel (Graphplan-based) planners, like IPP (by Köhler), Blackbox (by Kautz and Selman), Graphplan (by Blum and Furst), Stan (by Fox and Long) perform well on Logistics. Note, that heuristic search planners, such as (parallel) HSP2 with an IDA* like search engine loose their perfomance gains when optimality has to be preserved.

With *Pure* BDDA* and the FF-Heuristic, however, we can solve 11 of the 30 problem instances [20]. The dauting problem is that – due to the large minimized encoding size of the problems – the transition function becomes too large to be build. Therefore, the Logistics benchmark suite in the *Blackbox* distribution and in AIPS-2000 scale better. In AIPS-2000 we can solve the entire first set of problems with heuristic sybolic search and Table 2 visulizes the effect of *Pure* BDDA* for the *Logistics* suite of the *Blackbox* distribution, in which all 30 problems have encodings of less than 100 bits. We measured the time, and the length of the found solution. H^{HSP}_{add} and H^{HSP}_{max} abbreviate *Pure* BDDA* search according to the *add* and the *max* relation in the HSP-heuristic, respectively. H^{FF}_{add} and H^{FF}_{max} are defined analogously. The depicted times are not containing the efforts for determining the heuristic functions, which takes about a few seconds for each problem. Obviously, searching with the *max*-Heuristic achieves a better solution quality, but on the other hand it takes by far more time. The data indicates that on average the *FF-Heuristic* leads to shorter solutions and to smaller execution times. This was expected, since the average heuristic value per fluent in H^{FF} is larger than in H^{HSP}, e.g. in the first problem it increases from 2.96 to 4.43 and on the whole set we measured an average increase of 41.25 % of the heuristic estimate.

The backward search component - here applied in the regression space (thus corresponding to forward search in progression space) is used as a breadth-first *target enlargement*. With higher search tree depths this approach definitely profits from the symbolic representation of states.

In *Pure* BDDA* forward simplification is used to avoid recurrences in the set of expanded states. However, if the set of reachable states from the first bucket in the priority queue returns with failure, we are not done, since the set of goal states according to the minimal heuristic value may not be reachable.

	BFS		H^{HSP}_{add}		H^{HSP}_{max}		H^{FF}_{add}		H^{FF}_{max}	
1	25	0.66	30	0.06	25	1.05	30	0.92	25	0.49
2	24	121	27	5.33	24	129	31	1.27	26	3.52
3	-	-	29	3.30	26	35.98	28	1.18	26	30.22
4	-	-	59	6.53	52	37.10	59	3.49	52	22.74
5	-	-	52	5.64	42	4.56	51	3.11	43	3.41
6	42	72	63	7.22	51	67.18	64	2.45	52	11.37
7	-	-	83	14.89	-	-	80	11.87	-	-
8	-	-	84	19.14	-	-	80	15.05	-	-
9	-	-	84	13.07	-	-	80	8.94	-	-
10	-	-	47	13.93	40	484	45	8.15	40	421
11	-	-	54	10.10	-	-	52	7.30	-	-
12	-	-	37	1.19	-	-	36	3.90	-	-
13	-	-	77	15.18	-	-	78	9.89	-	-
14	-	-	74	18.58	-	-	83	13.36	-	-
15	-	-	64	17.16	-	-	68	10.08	-	-
16	39	580	49	7.19	41	4.64	46	2.78	40	1.73
17	43	277	51	9.97	43	3.91	50	2.60	43	3.38
18	-	-	56	21.53	-	-	54	15.76	-	-
19	-	-	53	12.85	-	-	57	8.01	-	-
20	-	-	101	20.42	-	-	95	13.58	-	-
21	-	-	73	16.16	-	-	69	10.47	-	-
22	-	-	94	18.45	-	-	87	14.54	-	-
23	-	-	72	13.95	-	-	71	10.81	-	-
24	-	-	79	14.18	-	-	75	9.50	-	-
25	-	-	73	14.81	-	-	66	9.03	-	-
26	-	-	60	14.23	-	-	61	9.35	-	-
27	-	-	81	15.31	-	-	80	12.72	-	-
28	-	-	87	27.15	-	-	89	23.74	-	-
29	-	-	51	21.58	-	-	52	16.70	-	-
30	-	-	59	13.41	-	-	59	9.61	-	-

Table 2. Searching the Logistics domain with *Pure BDDA**.

5.3 Model Checking Domains

The model checking problem determines whether a formula is true in a concrete model and is based on the following issues (cf. F. Giunchiglia and P. Taverso [25]):

1. A domain of interest (e.g, a computer program or a reactive system) is described by a formal model.
2. A desired property of finite domain (e.g. a specification of a program, a safety requirement for a reactive system) is described by a formula typically using temporal logic.
3. The fact that a domain satisfies a desired property (e.g. the fact that a program meets its specification, e. g. that a reactive system never ends up in a dangerous state) is determined by checking whether or not the formula is true in the initial state of the model.

The crucial observation is that exploring (deterministic or non-deterministic) planning problem spaces is in fact a model checking problem. In model checking the assumed structure is described as a Kripke structure (W, W_0, T, L), where W is the set of states, W_0 the set of initial states, T the transition relation and L a labeling function that assigns to each state the set of atomic propositions which evaluate to *true* in this state.

The properties are usually stated in a temporal formalism like linear time logic LTL (used in SPIN) or branching time logic CTL eventually enhanced with fairness constraints (used in PVS and SMV). In practice, however, the characteristics people mainly try to verify are simple safety properties expressible in all of the logics mentioned above. They can be checked through a simple calculation of all reachable states. An iterative calculation of Boolean expressions has to be performed to verify the formula *EF Goal* in the temporal logic CTL which is dual to the verification of *AG* $\neg$*Goal*. The computation of a (minimal) witness delivers a solution. Cimatti, Roveri and Traverso present BDD-based planning approaches capable of dealing with non-deterministic domains [12, 23]. Due to the non-determinism the authors refer to plans as complete state action tables. Therefore, actions are included in the transition relation, resulting in a representation of the form $T(\alpha, x', x)$. The concept of *strong cyclic plans* turns out to check the formula *AGEF Goal* [16] which expresses that from each state on a path a goal state is eventually reachable.

When using BDDA* for model checking safety properties it turned out that it is not a good choice to omit the set *Closed*. In difference to A*, however, the length of the minimal path to each state is not stored. The closest corresponding single-state space algorithm

is IDA* with transposition tables [45]. Unfortunately, even for optimistic heuristics it is necessary to memorize the corresponding path length to guarantee admissibility. However, one can omit this additional information when only *consistent* heuristics are considered. In this case the resulting cost-function is monotone. Fortunately, we found a refinement strategy to devise consistent heuristics for hardware verification [44].

In our experiments we used the μ-calculus [38] model checker μcke [1] which accepts full μ-calculus as its input language[3]. The *while*-loop of BDDA* can be converted into a least fixpoint. As it is not possible to change the two sets (*Open, Closed*) in the body of one fixpoint the *Closed* set is simulated by one slot in the BDD for *Open*. Another difficulty is that the function for *Open* is not monotone because states are deleted after they have been expanded. Monotonicity is a sufficient criterion to guarantee the existence of fixpoints. Therefore, the function for *Open* is not a syntactic correct μ-calculus formula but as the termination of the algorithm is guaranteed by the monotonicity of the *Closed* set the standard algorithm for the calculation of μ-calculus fixpoints can be applied nevertheless. Unfortunately, we cannot take advantage of a special BDD operation to determine the minimal costs in this case. These calculations have to be simulated by standard operations leading to some unnecessary overhead that in the visible future has to be avoided in a more customized implementation.

For the evaluation of our approach we use the example of the tree-arbiter a mechanism for distributed mutual exclusion: $2n$ users want to use a resource which is available only once and the tree-arbiter manages the requests and acknowledges avoiding a simultaneous access of two different users. The tree-arbiter consists of $2n-1$ modules of the same structure such that it is easy to scale the example. Since we focus on error detection we experiment with an earlier incorrect version published in [18] using an interleaving model.

	BFS			BDDA*		
n	it	max nodes	time	it	max nodes	time
15	30	991374	46s	127	5715484	288s
17	42	18937458	3912s	157	7954251	476s
19	44	22461024	6047s	157	8789341	540s
21	44	26843514	24626s (9)	157	9097823	530s
23	$>$40	-	$>$17000s	157	9548269	516s
25	-	-	-	169	21561058	1370s
27	-	-	-	169	25165795	1818s (1)

Table 3. Tree-arbiter: In parenthesis the number of garbage collections is given.

As the algorithm for the automatic construction of the heuristic has not yet been implemented and since the number of different error-cases increases very fast with the size of the tree-arbiter we searched for the detection of a special error. Table 3 shows the results in comparison with a classical forward breadth first search. To guarantee the fairness of the comparison the search is terminated at the time when the first error state has been encountered. The depth, which can be chosen by the user, denotes the quality and complexity of the automatically constructed heuristic depending on the transition relation and the error specification.

For the tree-arbiter with 15 modules or less the traditional approach is faster and less memory consuming, but for larger systems its time and memory efficiency decreases very fast. On the other hand, the heuristic approach found errors even in large systems, since its memory and time requirements increase more slowly. For the tree-arbiter with 23 modules the error could not be found with breadth-first-search. Already for the version with 21 modules 9 garbage collections were necessary not to exceed the memory limitations, whereas the first garbage collection with the heuristic method had to be invoked at a system of 27 modules. For the tree-arbiter with 27 modules we also experimented with the heuristic. When we double its values the heuristic fails to be optimistic, but the error detection could be carried through avoiding any garbage collections. Moreover, although more than three times more iterations were necessary only about 8% more time was consumed which indicates that it can be efficient to perform many iterations treating small sets of states instead of few iterations treating large sets. This also illustrates that there is much room for further research in refinements to the heuristic.

	BFS			BDDA*			
	size	max nodes	time	depth	it	max nodes	time
6	23	26843514	5864s (5)	6v	35	29036025	2207s (4)
				6	53	25165795	1009s (1)
				7	53	25159862	813s (0)

Table 4. Asynchronous DME: In parenthesis the number of garbage collections is given.

The second example used for the evaluation of our approach is the asynchronous DME. Like the tree-arbiter it consists of n identical modules and it is also a mechanism for distributed mutual exclusion. The modules are arranged in a ring structure whereas the modules of the tree-arbiter form a pyramid. In this case we also experimented with the set *Closed* and it turns out that it was more efficient to use the original BDDA*-algorithm. For this variation only small changes in the calculation of *Open* are necessary. Like in the previous example the results in Table 4 show that the heuristic approach is more memory efficient and less time-consuming. The first experiment in the table uses the set *Closed* that was omitted in the other experiments since this turned out to be more time and memory efficient.

6 Conclusion and Outlook

Symbolic breadth first search and BDDA* have been applied to the areas *search* [21] and *model checking* [44]. The experiments in *(heuristic) search* indicate the potential power of the symbolic exploration technique (in Sokoban) and the lower bound information (in the Fifteen Puzzle). In *model checking* we encounter a real-world problem of finding errors in hardware devices. BDD sizes of 25 million nodes reveal that even with symbolic representations we operate at the limit of main memory. However, the presented study of domain independent STRIPS-*planning* proves the generality of BDDA*.

The presented directed BDD-based search techniques bridge the gap between heuristic search planners and symbolic methods. Especially the newly contributed *Pure* BDDA* algorithm and the FF heuristic seem very promising to be studied in more detail and to be evaluated in other application areas. All applied heuristics are not as informative as their original, but, nevertheless, lead to good results. Together with the wide range of applicability through the generality of the presented approach, we conclude that on the same heuristic information a symbolic planner is competitive with a single state one if at least moderate-sized sets of states have to be explored.

Finally there is lot of work to be done in future. For example, some BDD refinements (such as transititon function splitting) should be implemented. Further on, we have to develop a BDD exploration

[3] All data in this section has been produced on a Unix-Workstation (Sun Ultra 1/512 MByte).

algorithm that yields optimal parallel plans. Kautz and Selman have shown, how this can be achieved in case of SATPLAN and Haslum and Geffner have presented a first solution to the problem for HSP. The most intense research will focus on generalization to the planning language (such as non-determinism), where the advantage of BDD-based planning compared to single-state exploration is more apparent. For example BDD generalisations to Markov decision process planning for the (single-state) *general planning tool* GPT by Geffner and Bonet are desireable [5].

When introducing ressources, hybrid approaches with integer programming become an apparent issue. Three different approaches can be found. Walser and Kautz integrate the concept of numbers within the propositional setting and therefore extend the languages [34]. However, the new formalism can handle plans with ressources, action costs and complex objective functions. Vossen et. al present a domain independent translation of planning problems into integer programs [46]. As a drawback the efficiency of other system has not been gained. Bockmayr and Dimopoulos integrate some domain specific knowledge to the setting of propositional planning [3].

Acknowledgment I thank F. Reffel and M. Helmert for their cooperation concerning this research. The work is supported by DFG in a project entitled *Heuristic Search and Its Application in Protocol Verification*.

REFERENCES

[1] Armin Biere, `μcke - efficient μ-calculus model checking', in *Computer Aided Verification*, pp. 468–471, (1997).
[2] A. Blum and M. L. Furst, `Fast planning through planning graph analysis', in *IJCAI*, pp. 1636–1642, (1995).
[3] A. Bockmayr and Y. Dimopoulos, `Integer programs and valid inequalities for planning problems', pp. 241–253.
[4] B. Bonet and H. Geffner, `Planning as heuristic search: New results', in *ECP*, pp. 359–371, (1999).
[5] B. Bonet and H. Geffner, `Planning with incomplete information as heuristic search in belief space', in *AIPS*, pp. 52–61, (2000).
[6] B. Bonet, G. Loerincs, and H. Geffner, `A robust and fast action selection mechanism for planning', in *AAAI*, pp. 714–719, (1997).
[7] R. E. Bryant, `Symbolic manipulation of boolean functions using a graphical representation', in *DAC*, pp. 688–694, (1985).
[8] R. E. Bryant, `Graph-based algorithms for boolean function manipulation', *IEEE Transaction on Computers*, **35**, 677–691, (1986).
[9] J. R. Burch, E. M.Clarke, K. L. McMillian, and J. Hwang, `Symbolic model checking: 10^{20} states and beyond', *Information and Computation*, **98**(2), 142–170, (1992).
[10] A. Cimatti, E. Giunchiglia, F. Giunchiglia, and P. Traverso, `Planning via model checking: A decision procedure for AR', in *ECP*, (1997).
[11] A. Cimatti and M. Roveri, `Conformant planning via model checking', in *ECP*, pp. 21–33, (1999).
[12] A. Cimatti, M. Roveri, and P. Traverso, `Automatic OBDD-based generation of universal plans in non-deterministic domains', in *AAAI*, pp. 875–881, (1998).
[13] T. H. Cormen, C. E. Leiserson, and R. L. Rivest, *Introduction to Algorithms*, The MIT Press, 1990.
[14] O. Coudert, C. Berthet, and J.C. Madre, `Verification of synchronous sequential machines using symbolic execution', in *Automatic Verification Methods for Finite State Machines*, pp. 365–373, (1989).
[15] J. C. Culberson and J. Schaeffer, `Searching with pattern databases', in *CSCSI*, pp. 402–416, (1996).
[16] M. Daniele, P. Traverso, and M. Y. Vardi, `Strong cyclic planning revisited', in *ECP*, pp. 34–46, (1999).
[17] E. W. Dijkstra, `A note on two problems in connection with graphs.', *Numerische Mathematik*, **1**, 269–271, (1959).
[18] D.L. Dill, *Trace Theory for Automatic Hierarchical Verification of Speed-Independent Circuits*, An ACM Distinguished Dissertation, The MIT Press, 1988.
[19] S. Edelkamp and M. Helmert, `Exhibiting knowledge in planning problems to minimize state encoding length', in *ECP*, pp. 135–147, (1999).
[20] S. Edelkamp and M. Helmert, `On the implementation of Mips', in *AIPS-Workshop on Model-Theoretic Approaches to Planning*, pp. 18–25, (2000).
[21] S. Edelkamp and F. Reffel, `OBDDs in heuristic search', in *KI*, pp. 81–92, (1998).
[22] S. Edelkamp and F. Reffel, `Deterministic state space planning with BDDs', in *ECP*, pp. 381–382, (1999).
[23] P. Ferraris and E. Guinchiglia, `Planning as satisfiability in simple nondeterministc domains', in *AIPS-Workshop on Model-Theoretic Approaches to Planning*, pp. 10–17, (2000).
[24] M. P. Fourman, `Propositional planning', in *AIPS-Workshop on Model-Theoretic Approaches to Planning*, pp. 10–17, (2000).
[25] F. Giunchiglia and P. Traverso, `Planning as model checking', in *ECP*, pp. 1–19, (1999).
[26] P. E. Hart, N. J. Nilsson, and B. Raphael, `A formal basis for heuristic determination of minimum path cost', *IEEE Transaction on SSC*, **4**, 100, (1968).
[27] P. Haslum and H. Geffner, `Admissible heuristics for optimal planning', in *AIPS*, pp. 140–149, (2000).
[28] J. Hoffmann, `A heuristic for domain independent planning and its use in an enforced hill climbing algorithm', Technical report, Computer Science Department, Freiburg, (2000). http://www.informatik.uni-freiburg.de/tr/133.
[29] S. Holldobler and H.-P. Stör, `Solving the entailment problem in the fluent calculus using binary decision diagrams', in *AIPS-Workshop on Model-Theoretic Approaches to Planning*, pp. 32–39, (2000).
[30] R. M. Jensen and M..M. Veloso, `OBDD-based deterministic planning using the UMOP planning framework', in *AIPS-Workshop on Model-Theoretic Approaches to Planning*, pp. 26–31, (2000).
[31] A. Junghanns, *Pushing the Limits: New Developments in Single-Agent Search*, Ph.D. dissertation, University of Alberta, 1999.
[32] H. Kautz and B. Selman, `Pushing the envelope: Planning propositional logic, and stochastic search', in *AAAI*, pp. 1194–1201, (1996).
[33] H. Kautz and B. Selman, `Unifying SAT-based and Graph-based planning', in *IJCAI*, pp. 318–325, (1999).
[34] H. Kautz and J. Walser, `State-space planning by integer optimization', in *AAAI*, (1999).
[35] J. Koehler and J. Hoffmann, `On reasonable and forced goal orderings and their use in an agenda-driven planning algorithm', *JAIR*, **13**(1), (2000). To appear.
[36] R. E. Korf, `Finding optimal solutions to Rubik's cube using pattern databases', in *AAAI*, pp. 700–705, (1997).
[37] R. E. Korf and L. A. Taylor, `Finding optimal solutions to the twenty-four puzzle', in *AAAI*, pp. 1202–1207, (1996).
[38] D. Kozen, `Results on the propositional μ-calculus', *Theoretical Computer Science*, **27**, 333–354, (1983).
[39] D. Long and M. Fox, `Automatic synthesis and use of generic types in planning', in *AIPS*, pp. 196–205, (2000).
[40] D. McDermott, `A heuristic estimator for means-ends analysis in planning', in *AIPS*, pp. 142–149, (1996).
[41] X. L. Nguyen and S. Kambhampati, `Extracting effective and admissible state space heuristics from the planning graph'. To appear.
[42] J. Pearl, *Heuristics*, Addison-Wesley, 1985.
[43] I. Refanidis and I. Vlahavas, `A domain independent heuristic for STRIPS worlds based on greedy regression tables', in *ECP*, pp. 346–358, (1999).
[44] F. Reffel and S. Edelkamp, `Error detection with directed symbolic model checking', in *FM*, pp. 195–211, (1999).
[45] A. Reinefeld and T. A. Marsland, `Enhanced iterative-deepening search', *IEEE Transactions on Pattern Analysis and Machine Intelligence*, **16**(7), 701–710, (1994).
[46] T. Vossen, M. Ball, A. Lotem, and D. Nau, `On the use of integer programming models in AI planning', in *IJCAI*, (1999).

Planning with tokens :

an approach between satisfaction and optimization

Patrick Fabiani and **Yannick Meiller**[1]

Abstract. This paper presents an initial approach in an attempt to integrate powerful propositional planning tools within a general planning process possibly involving both numerical and symbolic uncertainties. Recently developed propositional approaches to planning allow to build and search efficiently a planning graph. Yet, handling uncertainty and numerical optimization criteria within such frameworks remains difficult. A potentially powerful solution is to combine symbolic reasoning tools with decision theoretic representations. A first proposal is to begin with a colored Petri net representation of the propositional planning domain. Sample formalizations of planning problems and subsequent implementations are presented together with the propagation mechanisms with tokens. The plan is output as a Petri net, so that all information about dependency relations among actions is preserved. The preliminary results on deterministic planning problems seem encouraging. Besides, useful information concerning the planning process, uncertainty or risk can then be attached to the colored tokens and thus travel throughout the planning graph. This is meant to be later exploited for optimization.

1 Introduction

The introduction of sensory or modeling uncertainties in the classical planning paradigm leads to practical difficulties both at representation and solution level [13]. Yet, planning under uncertainty is a key issue in research about agents with autonomous decision making abilities. The topics addressed in [6] easily extend to the general case of autonomous agents which have to deal with rich domains while having uncertain or incomplete models of their environment, possibly involving partial observability or partial predictability. Recent work about a problem of dynamic robot motion planning under uncertainty [7] has shown how computational geometry could be combined to game theory in order to reduce the size of the search space. Though dealing with uncertainty is not the chief topic addressed here, this paper presents an initial step in order to generalize this approach and combine symbolic reasoning with decision-theoretic tools. The proposed approach consist in first designing an efficient classical planner borrowing ideas from Graphplan, to be later extended to allow planning under uncertainty. The present paper is organized in three sections. In section § 2 we first relate this work to other approaches and explain our motivations to use tokens with a discussion about optimization issues. In section § 3 we introduce Petri nets and tokens and show how this formalism can be related to our needs. In section § 4 we focus on the mechanisms of planning with tokens in classical domains, eventually providing perspectives on future work on both classical planning and planning with uncertainty.

[1] ONERA-CERT / DCSD, 2 av. Edouard Belin, F-31055 Toulouse cedex, email: fabiani@cert.fr, meiller@cert.fr

2 Related work

2.1 Combining decision theoretic and classical planning

The theory of games and decision [17, 16] provides an attractive framework for decision making under uncertainty, within which the study of automated sequential decision making under uncertainty can be usefully embedded. Different approaches to decision-theoretic planning have been studied and developed in various domains : see [14] for robot motion planning problems and [4] for a discussion about recent issues about decision-theoretic planning and more specifically around the MDP framework.

On the other hand, solving MDPs or stochastic dynamic programming problems of quite reasonable size remains a task of high computational complexity [15]. The curse of dimensionality appears when partial observability is added (POMDPs) : then, the dimension of the search space is equal to the size of the underlying state space. Algorithms have been developed that can efficiently solve decision-theoretic planning problems of reasonable size : in particular, [4] reviews a number of approaches developed to prune as many branches as possible in the search graph.

More specifically, the work in [3] is remarkable as proposing to reuse ideas from Graphplan [1] for reachability analysis in solving MDPs. Yet, it may not always be easy to draw an adapted MDP-like discretization of the state space from the initial problem definition : in [7] for example, the workspaces of the pursuer and the target robots are both systematically discretized into 1500 possible free states each (to be fine enough). Assuming perfect localization in real time for both robots, and focusing on couples of initial positions for which the pursuer can see the target (an average 40 possibilities), the MDP for the tracking problem with uncertain moves but perfect localization information would have a state space of size 60000. Now with localization uncertainties, the corresponding POMDP has a dimension 60000 : there would be a need for a tailored hierarchical decomposition rather than a systematic discretization. The curse of dimensionality is hopefully circumvented in [7] thanks to Computational Geometry tools, which seems a good idea to generalize whenever equivalent powerful tools exist.

This again, leads to the idea that classical planning methods could be somewhat adapted so as to take uncertainty measures into account at planning time, or alternatively, as already proposed by C. Boutilier [3], in order to build an appropriate search graph and allow a subsequent decision process to deal with it properly. For instance, *PGraphplan* in [2] takes probabilistic actions into account and *Sensory Graphplan* [21] handles uncertainty about the initial state. [10] presents another attempt to extend classical planning methods to dynamic and changing environments. Yet, most of the difficulties and limitations raised by S. Kambhampati in [13] still apply.

2.2 Planning with uncertainties, optimization and forward search

The first point we want to make here is that in classical planning, the problem is one of satisfaction of a sequence of transition conditions leading to a goal termination condition. By contrast, dealing with uncertainties in planning, like in decision-theoretic planning, requires optimization capabilities. This actually is one major difficulty for classical planning algorithms to adapt to problems with uncertainty.

For instance, [2] describes an adaptation of Graphplan for doing contingent planning, considering probabilistic actions - Pgraphplan. Pgraphplan builds the graph basically the same way as Graphplan. However, Graphplan considers that both instances p_1 and p_2 of a same proposition appearing in two separate states s_1 and s_2 are equivalent. For that reason, this proposition is introduced only once in a level of the graph - this is the basis of disjunctive planning. By contrast, this assumption is not valid anymore in Pgraphplan's plan-graph. Indeed, the reachability probabilities concerning s_1 and s_2 may be different, so that p_1, for example, may have a higher probability of reachability. Since we are doing contingent planning, it is important to keep this distinction between p_1 and p_2 in order to make the search for a plan easier. Unfortunately, Pgraphplan does not keep it because it builds the graph basically the same way as Graphplan.

As a consequence, in order to extract a plan, the plan graph has to be searched forward instead of backward. This prevents it from taking advantage of most of the speed-ups of classical Graphplan, because these are related to the coupling between forward constraint propagation and backward search. As pointed out in [2], searching the Pgraphplan's plan-graph backward is much more difficult than it is in classical Graphplan.

Similarly in [10], the author try to reuse Graphplan's ideas for a combination of contingent and conformant planning in an uncertain environment. In their approach though, the actual computation of uncertainty levels requires an unreasonable series of backward and forward computation phases.

The same problem occurs about utility functions or rewards : indeed, a given state can be reached from different trajectories with different utility or cost and one would need to keep some track of it in order to do some optimization. Besides, two different states, with two different utility values may share some common partial description. What value must be assigned to the node corresponding to this partial state when no distinction is made between the underlying states, or sets of states.

2.3 States or features ?

Most of the time, the probabilities, costs, utilities, or expected utilities to be optimized can only be computed via a forward search in the state space : the optimization criteria depend upon the states (or sets of states in the best case) and transitions between those states (resp. sets of states) along the performed sequence of actions. These criteria generally cannot be computed from features of the state, like features described by propositions, which is a real problem for disjunctive planners like Graphplan.

Here is a simple optimization example to fix the ideas. It is adapted from the rocket benchmark domain to fit in this discussion. Consider one rocket and four planets : Venus, Mars, the Moon, and the Earth. Two packages - A and B - are on the Earth, along with the rocket. A has to be delivered on Mars and B on Venus, **minimizing** the cost of transportation. Costs are attached to interplanetary journeys and are summarized in figure 1. How could we adapt Grpahplan to find the best transportation plan ? Consider two possible trips of the rocket :

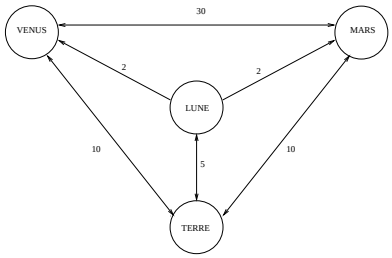


Figure 1. Interplanetary journeys have a cost. Here is an example.

1. Earth - Mars - Moon, $cost = 12$
2. Earth - Moon - (stayed on the Moon), $cost = 5$

While building the plan graph, we observe there are two ways of reaching the moon, with dramatically different cost values (suppose for example that each token wears a cost value that is updated when going through a FLY transition). A temptation is to keep the lowest cost token only. Now suppose while on Mars (in trip 1), the rocket unloaded package A. The higher cost of this trip is then totally justified. In fact, both these costs cannot be compared because they do not correspond to plans of same level of achievement with respect to the final goals to be reached.

To distinguish clearly between the two ways of reaching a node in the plan graph, we need to somewhat keep track of the followed *"trajectory"*, or the sequence of action. The fact is that for this specific domain, propagating tokens in a graph at planning time would allow the tokens to carry all the necessary information for the backward search algorithm to find the best cost afterward, thus naturally solving the problem.

Yet, generalizing this scheme would simply make our approach look very much like a forward search in the state space, and forward search in the state space is extremely costly. We would like to avoid it whenever possible.

2.4 Search space splitting and optimization

As a matter of fact, at each stage of the search space building process, some sets of (reachable) states may share interesting properties. They may have the same utility value - defining *utility regions* among states, like in [7] or they may be reachable with the same probability - defining *probability or risk* regions. In such cases, disjunctive planning, with its capability to manage sets of states (described as propositional features) may have an answer.

This is not always possible : when, on the contrary, the topology of these regions depends on the values assigned to intricate combinations of several features of the domain (thus, defining states), fully disjunctive planner are useless since they consider each feature independently. Intuitively, a nice solution should involve some "splitting" control based on a decomposition of the state space into regions, or sets of states, but in a way that has nothing to do with the way Graphplan handles features.

For instance, *Sensory Graphplan* [21] aims at handling uncertainty about the initial state. In the given example in [21], there are two possible hypotheses for the initial state and therefore, *Sensory Graphplan* generates two separate plan graphs, from each initial state. Each

graph is related to a possible world. The plan search looks for a plan valid in both worlds. In this case, both worlds are characterized by simple features, but doing so, the authors have introduced some splitting in the search space, based on an partition of the state space at the root of the search space. We would like to authorize this at any stage of the search space building process, whenever introducing such distinctions may happen to prove useful.

2.5 Controlling the search space splitting

For the rest of this paper, one says that the search space building stage *splits* the search space, when it splits the current set of reachable states in several sub-sets, and then continues independently from each sub-set. This notion is explained in more details in [12], in terms of planning approaches and potential plan sets.

For example, an *FSS* (*Forward State Space*) search type of approach does *full splitting*. Indeed, as soon as an action is introduced in a plan prefix - narrowing down the current set of potential plans - the resulting set is pushed in a new branch of the search tree (see fig. 2). On the contrary, a (disjunctive) Graphplan-like approach does *no splitting at all* : all the possible actions are introduced together, and the set of all the potential plan sets they entail is considered as a whole when continuing planning. This is the basis of disjunctive planning.

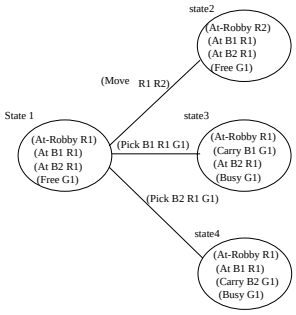


Figure 2. Here is the beginning of an FSS-search tree that would be developed for the gripper problem (with 2 balls and 1 gripper). Each node is a coherent state.

During the search space building stage, the state reachability may thus be more or less approximated. For example, Graphplan computes only pairwise mutex relations and therefore considers some states as reachable whereas they are not because of some three-wise constraint. In such a case, no plan can be found at backward search time, and a further extension of the search space is required.

The more splitting is used, the more accurate is this approximation. Doing full splitting insures exact reachability and optimality criteria computation. As a consequence, the checking stage in a deterministic goal oriented planning problem is trivial and classical Dynamic Programming tools can be applied directly if optimization has to be performed. With partial splitting, such criteria can at best be bounded, which may nevertheless prove useful.

The same way, the more splitting is used, the more narrow is the range of actions and propositions which are introduced. Indeed, we avoid some mistakes which may have led to the introduction of some actions that should be pruned. However, a higher level of splitting entails a higher level of duplication of work since each action may be introduced several times (from each subset of state) at each level. On the one hand, this makes the search stage easier - because part of the check is implicitly included in the classification, but on the other hand, this worsens the combinatorial aspects of the problem - because of the redundancy it entails.

2.6 Motivation for the use of Petri nets and tokens

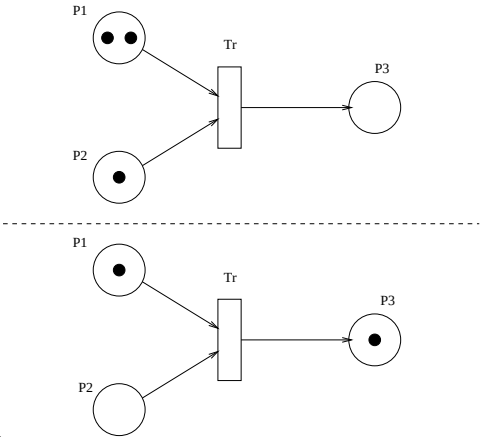
The first benefits we expect from Graphplan-like schemes is the two-stage process of forward search space building followed by backward search. On the other hand, some information may be drawn during the building phase that we don't want to lose for the search (and optimization) phase. The intuitive idea is that token propagation via Petri-like graphs should allow to keep track of some of this information and reuse it later, either during the search space building phase or during the backward search phase.

The token propagation mechanisms should support the optimization capabilities which are barely compatible with Graphplan's disjunctive planning : as a first contribution, we show in the following that we can provide token propagation mechanisms that allow us to perform both *Forward State Space* search and disjunctive planning *"à la Graphplan"*.

Planning with Petri nets is not a completely new idea [9] : it is here combined with recent research on Graphplan [12, 11]. In [8, 19], the expected advantages of such a combination are discussed in terms of action and perception planning for autonomous agents. More generally, it allows to attach useful information concerning the planning process, uncertainty or risk to tokens and then track or use this information throughout the planning graphs.

3 Tokens and Petri nets

3.1 From a STRIPS domain to a Petri net



chch

Figure 3. A simple Petri net, composed of a single transition Tr, and 3 places $P1$, $P2$, and $P3$. In the top figure, both $P1$ and $P2$ are marked by tokens (black dots), therefore the transition Tr can be triggered. On the bottom figure, the transition has been triggered : one token in each input place has been "consumed", and the output place $P3$ is marked by a token. Note that it remains a token in $P1$; however, since $P2$ is empty, the transition cannot be triggered. In more complex Petri nets, places can be connected to several transitions, as inputs or outputs.

A Petri net is composed of three types of elements : *places*, *transitions* and *tokens* (see fig.3). Places can be seen as token holders. They are connected to transitions as inputs or outputs. When a place contains one or more tokens, it is said to be *marked*. All of the places connected as input of a same transition must be marked before this transition can be triggered. When a transition is triggered, one token

of each of its input places is “consumed”, and all of its output places are marked. This new marking can allow other transitions to be triggered and so on. In *colored* Petri nets, tokens can be of different colors. In this case, tokens’colors add new constraints for determining whether a transition can be triggered or not, and of course, additionnal rules allow transitions to compute the colors of tokens marking output places.

It is pretty straightforward to represent a STRIPS domain using Petri nets [9]. First, consider totally instantiated operators. For each possible proposition, the Petri net contains a place. Transitions correspond to operators. They are connected to the places related to their preconditions as inputs, and to the ones related to their effects as output. Once the STRIPS domain has been translated in a Petri net (see fig. 4), a state is represented by the global marking of the net.

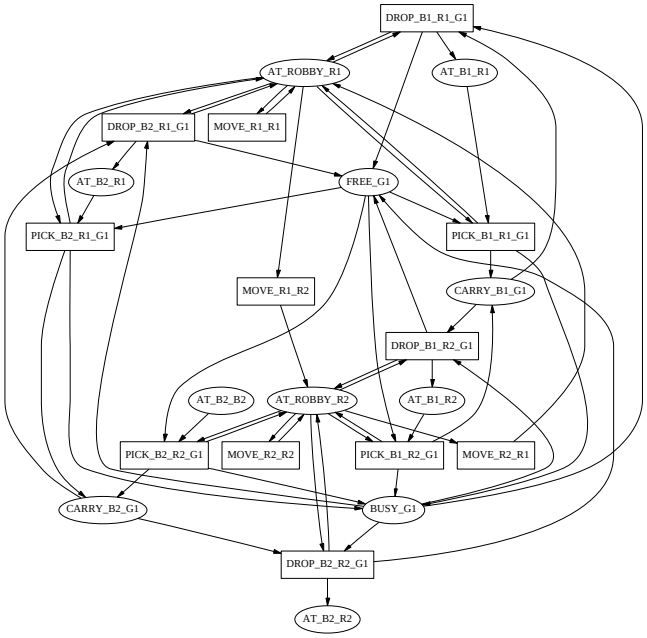


Figure 4. Petri net corresponding to an instance of a *gripper* domain. In this example, operators have been instantiated considering one gripper $G1$, two balls $B1$ and $B2$, and two rooms $R1$ and $R2$

When a place is marked by a token, the corresponding proposition is true (false otherwise, as in STRIPS). At a given time, the marking shows also all the transitions that can be triggered. This representation can be made more compact considering non-instantiated operators.

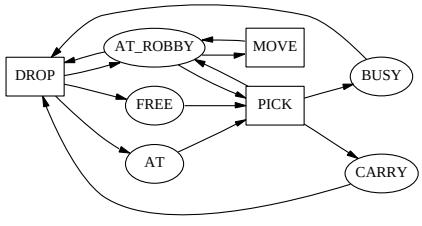


Figure 5. Petri net involving labels generated starting from a *gripper* domain (the same as the one of fig. 4). This graph, much more compact, represents non-instantiated operators. The pertinent instances will be brought using the tokens’labels.

3.2 From a PDDL domain to a colored Petri net

Let now each place correspond to a feature of the domain (see figure 5) : now each token has to hold a label on which is written an instantiation of the variables in the feature. From a propositional logic point of view, a place corresponds to a predicate, and the label of a token wears a particular binding. As a consequence, whether a transition can be triggered depends also on the labels of the tokens marking its preconditions. Reversely, the execution of a transition may not entail only a token motion but also some modification of their labels.

Practically, starting with a PDDL [18] description of a domain, its transcription into a Petri net proceeds automatically as follows. Each `:predicate` gives a place, and each `:action` a transition. Connecting to a transition places corresponding to preconditions or *positive effects* is obvious. *Negative effects* (i.e. when the operator destroys some of its preconditions) are naturally dealt with at the Petri net level, since a token marking a precondition place leaves it when the transition is triggered. Therefore, these bring no change to the net topology. A third type of effect has to be considered, we call them *implicit effects*. They encode the fact that some of the preconditions of the action remain true after its execution - i.e. propositions appearing in the precondition list without being in the *negative effect* list. These require additional links to be introduced in the network, in order to bring the tokens back to the places where they were before their transit through the transition (since these preconditions are still true). See fig. 6 for an example. [htbp]

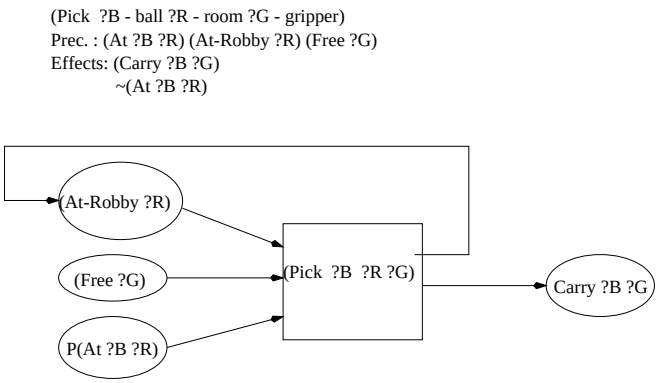


Figure 6. On top of the figure is the PDDL description of the operator $Pick$ (from the gripper domain). Below is its transcription as a Petri net transition. Note that there are two exiting links (on the right), whereas this operator has only one positive effect. This is because the top link - the one returning to At-Robby - is an implicit effect.

For each link in the Petri net, unification between variables of the proposition attached to the place and variables of the operator attached to the transition is completed and stored. This will make easier the computation related to the propagation of tokens through transitions (particularly when dealing with the modification of their labels).

Note that the Petri net obtained this way reflects exactly the domain as it is described in PDDL. Therefore, it is independent of any specific planning problem. Thus it needs to be built only once per domain.

4 Planning with tokens

4.1 Building the search space like Graphplan

Building the space of reachable states is done by propagating tokens from an initial marking, through transitions. The way Graphplan completes this kind of task (extending the planning graph) has proven its efficiency, and has displayed characteristics particularly attractive for planning [1, 13]. This section shows that the same can be done in our framework, inheriting by the way the same properties. Building the space of reachable states is done by propagating tokens from an initial marking, through transitions. Every place holds a list of token levels in which are recorded the successive markings. Therefore, a token is always considered within a given *token level* t of an item of the Petri net. It is linked to tokens in the $t-1$ *token level* of items of the net (showing where it comes from), and to tokens in the $t+1$ *token level* of items of the net (its potential destinations). This way, we can track the trajectory of a token during the propagation, and get the list of transitions it went through - i.e. the corresponding plan.

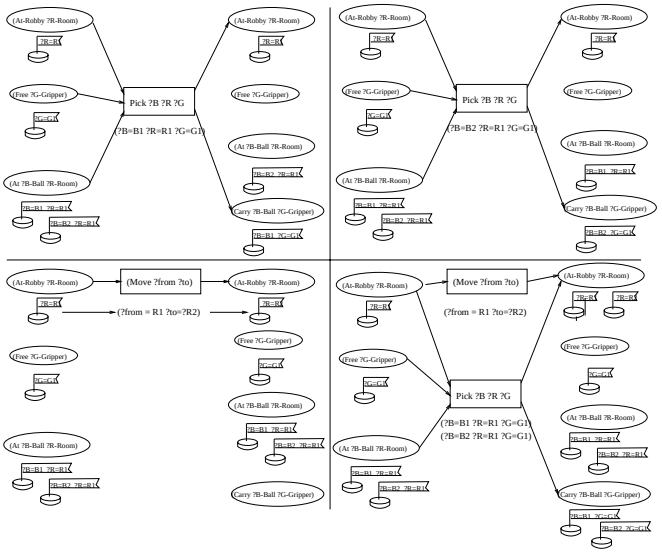


Figure 7. In every picture, the first column of places is the same, showing the marking at time t. The second column shows a possible marking at time t+1. The three first pictures correspond to the three transitions which can be triggered according to the current marking. The last one, on the bottom right hand corner, shows what we obtain with our approach : each possible transition has been triggered "virtually" so that the other ones could be triggered also.

At propagation time, as soon as the preconditions of a transition are marked, the transition is "virtually triggered". That is to say that even though tokens are sent to the next token level of the places corresponding to the positive and implicit effects of the transition, a copy of the current marking persists so that other possible propagations can be conducted (see fig. 7). As a consequence, at each step of the propagation, the global marking of the current token level corresponds to the union of all possible markings. Hence, it does not describe a single state but a set of reachable states.

As you can see in figure 8 the graph representing the tokens'connectivity is very similar to the one built by Graphplan.

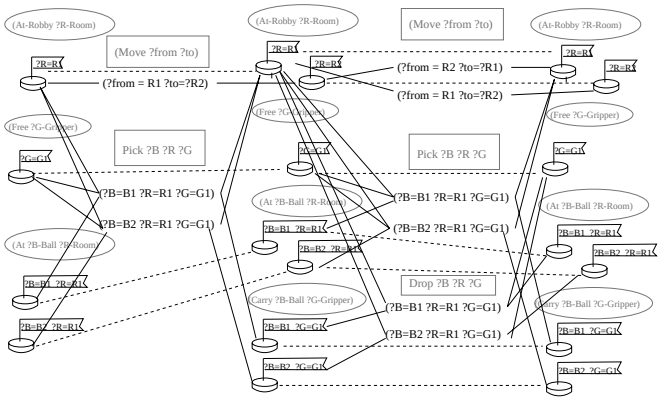


Figure 8. Each level shows the new token level of the places. In between, the transitions which caused the token motion are recorded. This plan graph is very much alike Graphplan's one.

So far, at each step, all the markings which can be deduced from the global marking of the net may include some unreachable ones. This is a common issue in disjunctive planning. In order to discard part of these impossible states, additional constraints are usually introduced : for instance, Graphplan computes *mutex relations*, and LCGP [5] computes *authorization relations*. In our system, mutex relations are computed, according to rules similar to Graphplan's ones (as stated in [1]). In terms of tokens, the *interference* rule (*"à la Graphplan"*) is as follows : if a transition T *"consumes"* a token, then it is mutex with all transitions using the same token (because T deleted one of their *preconditions*), and with all *other* transitions bringing an identical token (same label) in the same place (because T deleted one of their *positive effects*). In other words : two transitions cannot *"use"* the same token simultaneously, unless they both get and put back an identical token in a same place instantaneously (implicit effects). The *competing needs* rule (*"à la Graphplan"*) becomes : if there is a token t_1 triggering a transition, and a token t_2 triggering another transition, and if t_1 and t_2 are mutex, then these transitions are also mutex. Of course, the same thing could be done with *authorization* relations.

Similarly, among propositions : two tokens are mutex if every transition bringing the first one in its current place and giving it its current label is mutex with all the ones bringing the second one in its current place with its current label. Similarly to Graphplan, if two tokens marking precondition-places of a same transition are mutex, then the transition is not triggered.

The next section shows that some of the mutex relations need not to be explicitly computed, thanks to the use of colored tokens.

4.2 Building the search space slightly differently

The so far obtained planning graph is very similar to Graphplan's plan-graph, with the slight difference that each level is distributed among places and transitions. As a side effect, when searching for the particular instance of a proposition (to check whether an effect has already been introduced in the graph, for example), it is not necessary to search the whole list of propositions of the given level, but only the list of instances of the relevant place.

However, with the following rule, the use of colored tokens can allow to further avoid computing explicitly a number of *"permanent"*

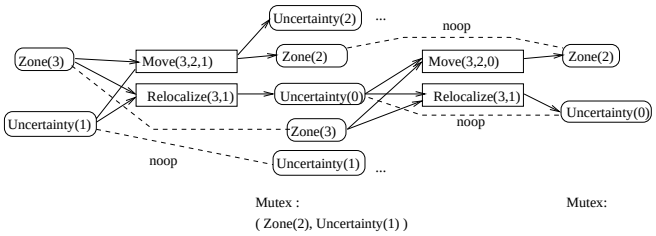


Figure 9. Example of mutex relation directly related to the planning problem. It disappears during graph construction. From the graph built for this mobile robotics problem, only a part relevant to our discussion is represented

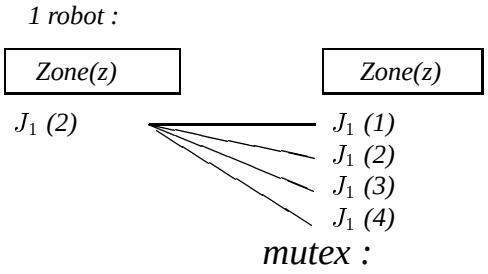


Figure 11. From zone 2, the robot can reach three other zones, or also remain where it is. Thanks to the use of colored tokens (here colors are represented by subscripts), no mutex needs to be expressed explicitly.

mutex relations between propositions carried by tokens of same color marking a same place (concurrent instantiations of a same feature) : *Two tokens of the same color (coming from the same past trajectory) marking a same place with two different labels are mutex.*

Indeed, two types of *mutex* relations are generated by Graphplan. Some of them are directly related to the planning problem itself, i.e. to the initial marking in the planning domain. They disappear after the plan-graph has been extended over some levels, and the range of reachable states consequently increased (see fig.9). Some other *mutex* relations, on the contrary, are *"permanent"*, or structural. Related to the domain itself, they will never disappear from the graph. They correspond to states which are strictly impossible, and therefore which can never be reached! Such *mutex* come from the fact that some features, or variables, cannot have more than one assignation at a given time, and in a given context. For example, a same robot cannot be concurrently in two different places (see fig. 10). In other words, some resources cannot be shared or some objects cannot be in different places at the same time. Comparable results are obtained by R.M. Simpson and T.L. McCluskey with their *Object-Graph planner* [20].

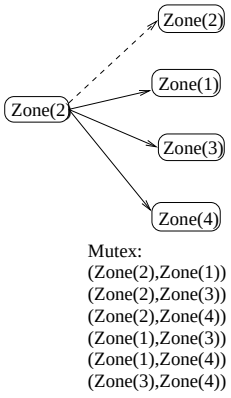


Figure 10. From zone 2, the mobile robot can reach 3 other zones, or also it can stay where it is. Graphplan is forced to introduce six permanent mutex relations.

Figure 11 shows the same situation as fig. 10 within our framework: No explicit mutex relation needs to be computed. They are all embedded in the token coloring.

As a more complete example, consider the possible motions of two robots *R1* and *R2* that can independently move in room *1*, *2*, *3* or *4*. Two tokens, denoted with different subscripts $_1$ and $_2$ are needed. Figure 12 shows the graph we get in this case. In the second level, place $InRoom(room, robot)$ is disjunctively marked by two sets of four tokens of same *"color"* (subscript $_1$ or $_2$ in this case). According to the rule, tokens of same subscript in the same level are all mutually exclusive with each other (a same robot cannot be simultaneously in different places!) but not with tokens of different subscript (the positions of the two robots are not dependent upon one another). There is a sort of parallel between the robot's non-ubiquity and the one of the token. In that sense, the tokens'colors are used here in order to represent the non-ubiquity of "objects" or "agents". In comparison to Graphplan, this rule leads to the computation of a smaller number of explicit *mutex* relations.

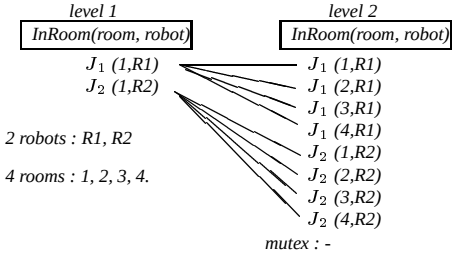


Figure 12. Both robots can reach three other rooms, or remain where they are. Thanks to token coloring, no mutex has to be expressed explicitly. Every position pair permitted by Graphplan is still permitted here.

The more numerous will be the variables not supporting multiple instantiations, and the literals which can be assigned to them, the more numerous will be the permanent *mutex*. Graphplan generate them explicitly, as regular *mutex* relations, making it more difficult to store and treat its data structures. We encode them implicitly through the use of colored tokens. These remarks may explain the speedup obtained in preliminary implementations and presented in table 1.

4.3 Handling colors - practically

Applying technically the above considerations is not always straightforward. Consider a transition with two preconditions and a single effect... which precondition token will give its color to the effect one - keeping in mind this will determine its potential mutex relations with other tokens marking the same place? Our concern being to

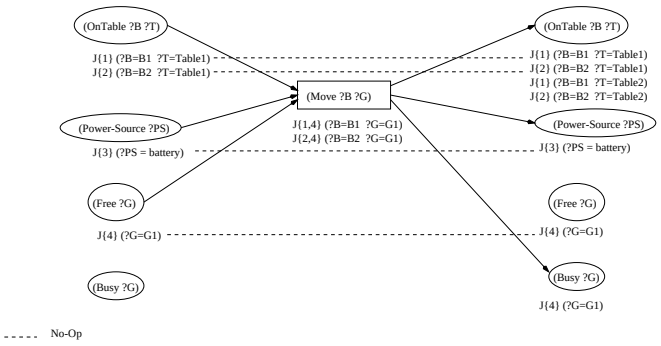


Figure 13. In the initial state, each token has a different color. As $Power - Source$ is an implicit effect, its color is not part of the action color (which therefore has only 2 components). In the second level, the third token of $OnTable$ wears the color 1 : this is the intersection between the action color ($\{1, 4\}$) and the set of colors marking this place in the initial state ($\{1, 2\}$). The token marking $Busy$ wears the color 4 because this is the only remaining component of the action color.

have a fully automatic planner, this choice must be made through a systematic method.

Firstly, here is a naive solution one could think of, based on the possible aggregation of colors. Let the colors be represented by sets of integers. In our example, say the first precondition is marked by a token of color C_1, and the second one by a token of color C_2, then the resulting color of the token marking the effect would be $C_1 \cup C_2$. For two tokens, of respective color C_i and C_j, marking a same place, if $C_i \cap C_j \neq \emptyset$ then there would be a mutex relation between both these tokens.

This does not work because the color treatment would not make any difference between tokens related to resources (sharable or reusable over time) and tokens related to objects (in the particular context of the transition being considered). Consider for example two tables with blocks on the first one, and two grippers used to move blocks from $Table1$ to $Table2$. The action $(MoveBlock?BGripper?G)$ requires the precondition $(OnTable?BTable_1)$ and $(Free?G)$. Of course, it is possible to move the block B_1 with the gripper G_1 and then to move the block B_2 with the same gripper G_1. This would cause the place $(OnTable?B?T)$ to be marked with two tokens, one carrying the label $(?B = B_1; ?T = Table2)$, and the other carrying the label $(?B = B_2; ?T = Table2)$. Unfortunately, both these tokens would have a color containing the color of the token marking $(FreeG_1)$, and therefore would be considered mutex in the "naive" plan graph whereas they are not. Here, the gripper plays the role of a resource, that is not sharable at one time, but usable several times in sequence.

Here is what is actually implemented in our system, which avoids these troubles. Let C_{action} be the union of the precondition colors of the considered action.

The color assigned to each effect is computed starting from C_{action}. In order to keep the tokens'coloring coherent with regard to the "objects" identification, some components of C_{action} have to be filtered out. Colors of tokens entering the transition (from preconditions) are related to some "objects" or "agents" ; the same consistency between "objects" and colors must be found among the tokens exiting the transition (towards effects). This is accomplished by maintaining a consistency between colors and places (instead of "objects" directly) and the best available reference concerning the coherent link between colors and places is the initial state. The assignment of effect colors is done as follows.

Preconditions being also implicit effects of the action are discarded from C_{action} (this takes care of sharable resources).

The effects of the action are split in two categories : places which were marked in the initial state (we know which colors are attached to them), and places which were not. For one of the first type, we keep from C_{action} only the components corresponding to the colors of tokens marking it in the initial state, and we remove these components from C_{action}. Effects of the second type will all receive the resulting C_{action}. Figure 13 shows this procedure on our simple example.

This approach makes a distinction, as much as possible, between resources and objects. The scenario we presented cannot occur, but still, the fact that the same ball cannot be on both tables at the same time remains encoded by the colors. Of course, in a description language like PDDL, as well as in natural language as a matter of fact, resources and objects using them can be confused sometimes - all the more as this distinction is highly context-dependent. For this reason, the number of mutex relations which will be encoded through the color manipulation depends upon both the way the domain is written and the initial state of the problem. Anyway, the planner remains complete since a lack of mutex relations is not harmful (only too many of them is harmful as far as it concerns completeness). Besides, regular explicit mutex checking is completed when token colors are not mutex, so that the backward search will have as many constraints as possible to be guided. Finally, note that in such a domain, colors of tokens marking places are single numbers - so that checking whether two tokens are mutex is easy and fast.

4.4 Obtaining a plan

Table 1. Comparison on the 6 balls GRIPPER domain with a code kindly provided by S. Kambhampati. Note that mutex relations not explicitly computed have been deducted thanks to tokens'colors.

	Graphplan + Csp tech.'s [11] 64 s 550 msec			Planning w/ tokens 4s 800 msec		
level	prop mut	action mut	records	prop mut	action mut	records
0	214	3858	0	136	2478	0
1	214	3858	1	136	2478	1
2	214	3858	45	136	2478	46
3	214	3858	254	136	2478	103
4	214	3858	1087	136	2478	204
5	214	3894	2724	136	2478	297
6	226	4206	5746	136	2478	781
7	250	4182	10774	136	2604	1261
8	322	3298	10237	148	1458	631
9	542	1466	0	112	680	0
10	220	176	0	140	121	0
11	0	0	0	0	0	0

As soon as the features of the *goals* are marked with the right tokens in the right places with the right labels, a backward search of the type of Graphplan's one can be applied to the graph generated by the token propagation in order to extract a valid plan - if any. A remarkable point is that the plan is output as a Petri net, so that all information about dependency relations among actions is preserved.

The presented planner based on token propagation mechanisms was implemented in Lisp on a Sun Sparc 10 Ultra and the GRIPPER domain has been automatically generated from a PDDL defini-

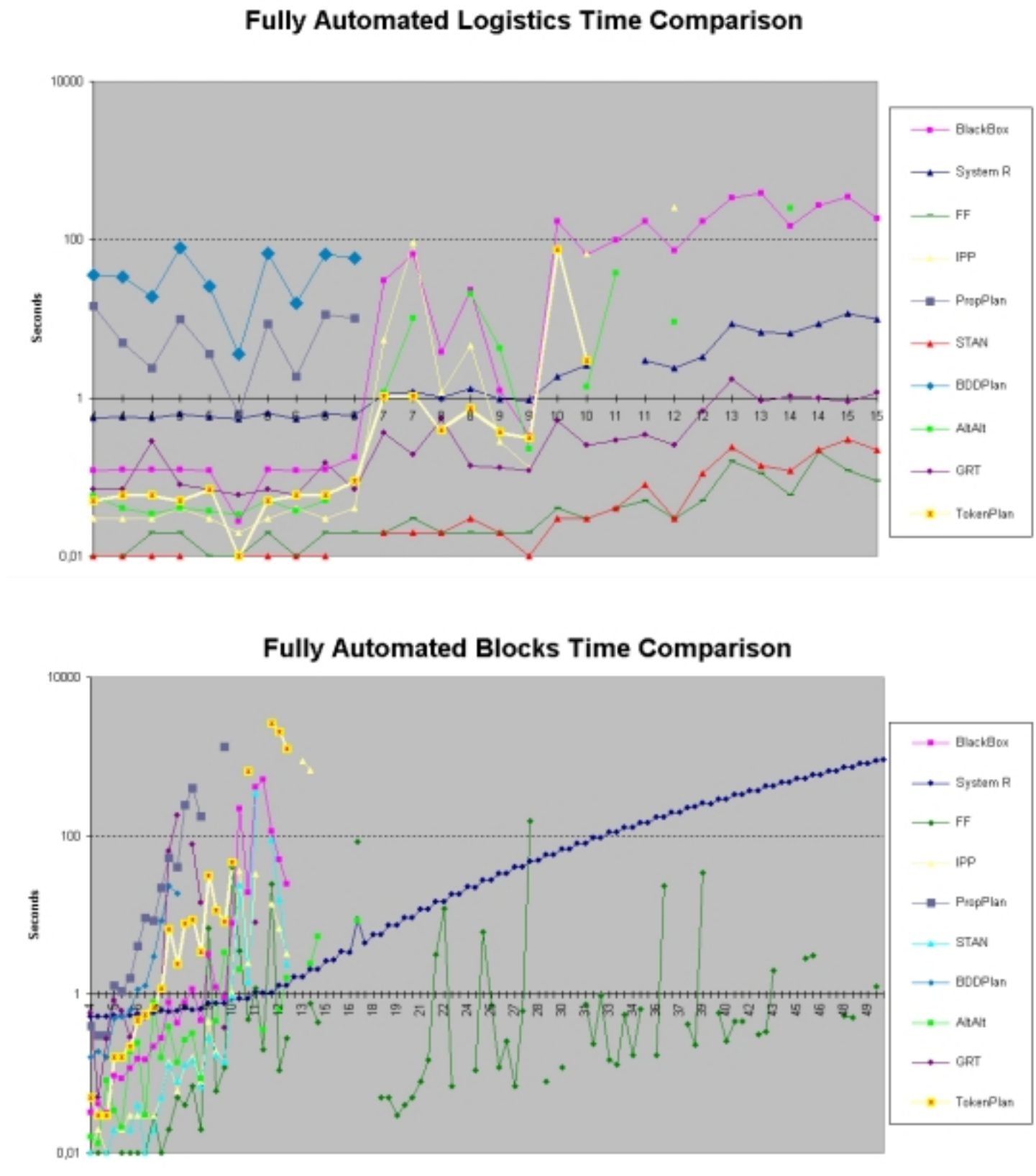


Figure 14. Both these graphs represent CPU-time performances of some of the planners participating to the AIPS'2000 planning competition. Missing points correspond to problems that could not be solved within the CPU-time limit (30 min).The top graph shows results on the Logistics domain, and the bottom one on the Blocks domain (abscissa values correspond there to the number of blocks involved in the problem)

tion file. A comparison was drawn with a Lisp code provided by S. Kambhampati.

The backward search algorithms are identical for both planners and are also from S. Kambhampati, who introduced in [11] the use of dynamic CSP techniques in order to boost the backward search phase. The table 1 gives both total planning time and for each level developed in the forward building phase, it compares the number of generated mutexes, among propositions or among actions, and sets of markings that are recorded as *"unreachable at that level"* after checking in the backward search phase.

The most remarkable feature in this comparison seems to be the difference in the number of mutex recorded by the two planners. The token planner need not record most of the *"permanent mutex"* classical Graphplan records and a subsequent simplification occurs both at propagation time, and backward search time : the number of constraints to check and the number of sets of markings to search are reduced consequently. There should be an increase in memory use, but it does not seem to be significant so far, contrary to the speedup.

Further experimentation has been completed during the AIPS'2000 Planning Competition. Our system entered it under the name *TokenPlan*. A sample of results is showed in figure 14, from data kindly provided by F. Bacchus, chair of this year competition. All the systems ran on a 500MHz Pentium III with 1GB of RAM. Some of them were implemented in C, C++, Lisp, or else in Java.

Despite these language differences, we can see TokenPlan is in the stream. Its performances are a bit better than BlackBox' (even though this one scales better), and roughly equivalent to IPP's. This shows the transcription of Graphplan in the Petri net framework is feasible and reasonable (no loss in performances is induced). The same way, as for other Graphplan based planner, it would be possible now to add features to Tokenplan in order to improve its performances (such as the computation of authorization relations instead of mutex relations [5] for instance).

4.5 Perspectives on search space splitting with token propagation

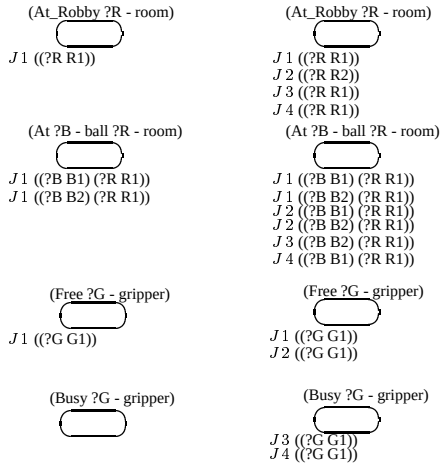


Figure 15. Here is the beginning of a full splitting search applied to the gripper problem in our framework. To make easier the comparison with fig. 2 we numbered the classes of tokens the same way we had numbered the state-nodes. Actions have been omitted for clarity.

Previous sections described how to build a fully disjunctive search space using token propagation. It can be rapidly checked that the same approach may as well generate full splitting by simply creating new classes of tokens each time a new state is reached : first consider tokens gathered in different classes (the class is represented here by an integer) - add the rule that in order to be triggered, a given transition must have all of its preconditions marked by tokens of the *same class* - add the rule that all the tokens marking the effects of a given transition are given a new class, which is not in use already. An example of the result is given in fig. 15. Some places may be marked by tokens of different classes wearing the same label. As an example, this is the case for $((?RR1))$ in fig. 15. This is the inherent redundancy of full splitting approaches.

As far as it concerns the example of [21], the splitting required to distinguish possible initial states can be achieved within our framework by introducing a different class of tokens for each possible world. Only one graph *per se* would be built.

Therefore, this may give the opportunity to duplicate only parts of the states that are actually different with respect to an optimization criterion : probability of reach and utility would be chief criteria for designing classes in a decision-theoretic planning problem such as the rocket one of figure 1.

Nevertheless, we have shown that we can provide token propagation mechanisms that allow us to perform both *Forward State Space* search and disjunctive planning *"à la Graphplan"*. Intermediate *"splitting"* strategies remain to be defined, implemented and optimized.

5 Conclusion

A lot of work remains to be done about the basic mechanisms of the proposed token-based planner and further validation and improvements are needed. Otherwise, it seems interesting to further study planning with intermediate levels of splitting as pointed out in [12, challenge4].

The basic idea is to be able to distinguish states or sets of states only when necessary. The framework proposed in this paper theoretically allows a large variety of splitting strategies, but the problem of controlling these strategies at planning time remains difficult. One solution would be to attach directly a sub-domain of the state space to the tokens. These sub-domains would correspond to a decomposition into regions of validity of properties related to utility functions or probability measures over the state space. Such a decomposition would be tailored for the computation of the stochastic optimality criteria just the same as regions of utility or reachability are computed geometrically in [7]. The advantage of tokens with that respect is that such a decomposition highly depends on the trajectory followed in the planning graph.

REFERENCES

[1] A. Blum and M. Furst, 'Fast planning through planning graph analysis', *AI*, (90), 281–300, (1997).

[2] A. Blum and J. Langford, 'Probabilistic planning in the graphplan framework', in *AIPS Workshop on 'Planning as Combinatorial Search'*, (1998).

[3] C. Boutilier, R.I. Brafman, and C. Geib, 'Structured reachability analysis for markov decision processes', in *UAI'98*, pp. 24–32, Madison, (jul. 1998).

[4] G. Boutilier, 'Decision-theoretic planning: Structural assumptions and computational leverage', *Journal of Artificial Intelligence Research*, **11**, 1–94, (1999).

[5] M. Cayrol, P. Regnier, and V. Vidal, 'New results about lcgp, a least committed graphplan', in *AIPS 2000*, pp. 273–282, Breckenridge, Colorado (USA), (2000).
[6] L. Dorst, M. van Lambalgen, and F. Voorbraak, eds. *Reasoning with Uncertainty in Robotics*. Springer, mar 1996.
[7] P. Fabiani and J.-C. Latombe, 'Dealing with geometric constraints in game-theoretic planning', in *IJCAI'99*, Stockholm, (aug. 1999).
[8] P. Fabiani and Y. Meiller, 'Théorie des jeux et planification pour le dilemme perception-action.', in *RFIA'2000*, PARIS, (2000). AFRIF-AFIA.
[9] F. Garcia, R. Mampey, and C. Barrouil, 'Génération de plans et révision des connaissances', in *RFIA'91*, Lyon-Villeurbanne, (novembre 1991).
[10] E. Guéré and R. Alami, 'Vers une planification en environnement dynamique', in *RFIA'2000*, PARIS, (2000). AFRIF-AFIA.
[11] S. Kambhampati, 'Planning graph as (dynamic) csp: Exploiting ebl, ddb and other csp techniques in graphplan', in *Ijcai'99*.
[12] S. Kambhampati, 'Challenges in bridging plan-synthesis paradigms', in *IJCAI'97*, (1997).
[13] S. Kambhampati, E. Lambrecht, and E. Parker, 'Understanding and extending graphplan', in *ECP'97*, (1997).
[14] S. M. Lavalle, *A Game-Theoretic Framework for Robot Motion Planning*, Electrical engineering, University of Illinois, Urbana-Champaign, 1995.
[15] M.L. Littman, T. Dean, and L.P. Kaelbling, 'On the complexity of solving markov decision processes', in *UAI'95*, pp. 394–402, Providence, (jul. 1995).
[16] L.J.Savage, *The Foundations of Statistics*, Dover, New York, 1972.
[17] R. D. Luce and H. Raiffa, *Games and Decisions*, John Wiley & Sons, New York, 1957.
[18] D. McDermott and AIPS-98 Planning Competition Committee, *PDDL -The Planning Domain Definition Language Version 1.2*, 1998.
[19] Y. Meiller and P. Fabiani, 'Perception-action dilemma at planning time : Getting the best out of game theory and classical planning', in *PLANSIG'99*, (1999).
[20] R.M. Simpson and T.L. McCluskey, 'An object-graph planning algorithm', in *PLANSIG'99*, (1999).
[21] D. S. Weld, A. R. Anderson, and D. E. Smith, 'Extending graphplan to handle uncertainty & sensing actions', in *AAAI'98*, (1998).

Scheduling in a Planning Environment

A. Garrido, M. A. Salido and **F. Barber**[1]

Abstract. In a real planning problem, there exists a set of constraints (both temporal constraints and resource usage constraints) which must be satisfied in order to obtain a feasible plan. This requires a scheduling process (after the planning process) which should guarantee the availability of resources and the satisfiability of all the problem constraints. Several approaches have been proposed to deal with planning and scheduling problems. However, these approaches have drawbacks which will be presented here. This paper deals with the main features of a scheduling process in an integrated architecture of planning and scheduling, where both processes work in a simultaneous way. Thus, the executability of each plan is guaranteed as it is being obtained by the planner. The planning process searches among alternative partial plans, where each one of them has its own ordering relations among actions, resource requirements, intermediate states, etc. Since these constraints are provided while the plan is being obtained, the proposed scheduling process should be able to manage them as they are being known. Thus, the scheduler should not obtain a solution after each new asserted constraint but rather it should only maintain the consistency among all the asserted constraints. In addition, the planner keeps track of several alternative open plans, which are suitable for being expanded in each moment. For this reason, the scheduler should maintain the effects of the constraints belonging to different plans that are being explored by the planner. Hence, both specific planning and scheduling optimisation criteria are used in order to improve the behaviour of the integrated system, its efficiency and the quality of the obtained plan.

1 INTRODUCTION

In a planning problem, actions usually require use of shared resources in order to be executed. Moreover, several temporal constraints should be satisfied during the plan execution: action durations, effect persistences, temporal constraints on problem states, due times, etc. In usual planning processes, resource usage and satisfiability of problem temporal constraints are not considered. Thus, planning systems obtain a plan as a partial or total ordered sequence of actions, and a later scheduler process should check the feasibility of the plan according to the available resources and problem constraints. Therefore, a correct plan may not be executable due to violation of some temporal constraint or unavailability of shared resources. Thus, a new plan should be obtained and there will be a loss of system performance.

[1] Dpto. Sistemas Informáticos y Computación, Universidad Politécnica de Valencia, Camino de Vera s/n 46071, Spain, email: {agarridot, msalido, fbarber}@dsic.upv.es

On the other hand, temporal planners can reason about metric constraints such as parcPLAN [11] and IxTeT [12]. These temporal planners deal with temporal data by means of an explicit representation of time managing qualitative (ordering relations commonly used in planning) and quantitative constraints (release times and durations used in scheduling). In a more integrated way, there are planning systems such as Tosca [4] and O-Plan [5, 7] which integrate both planning and scheduling processes in a single system. However, a drawback appears in these cases: it becomes difficult to determine when the system is planning or scheduling: "*it is easy to see that O-Plan works, but it is difficult to see why*" [1]. Since a specific process of planning or scheduling does not exist, it becomes difficult to determine certain optimisation criteria (which, moreover, can be integrated in a central module). Consequently, we agree planning and scheduling integration is necessary and these two processes should be performed simultaneously. However, we think these processes are different enough to be distinguished during their execution in the integrated system. Planning processes deal with what actions are going to be executed whereas scheduling processes deal with when these actions are going to be executed [8]. In another way, planning implies reasoning about actions and system states, and scheduling implies reasoning about actions, resources and time [27]. Therefore, it is not irrelevant to study both processes in a separate way in order to improve finally the performance of the integrated system [1]. Thus, an integrated system may obtain many benefits from both the planner and the scheduler, such as utilisation of shared heuristics, decrease the search space, performance improvements, etc. [17]. Nevertheless, this integration usually is not quite frequent due to the fact that it is neither easy nor intuitive: "*these systems do not integrate well*" [23].

This paper deals with the main features of a scheduling process in an integrated architecture for planning and scheduling [13]. In our system, the planner and the scheduler work simultaneously in an integrated way. Here, the scheduler guarantees the satisfiability of temporal constraints and resource availability for each partial plan as these plans are obtained by the planner. This way, the system recognises the invalid plan and this plan is immediately discarded. Furthermore, even though the plan is executable it may not be efficient enough or optimal. For this reason an alternative plan may be needed with the objective of reducing its cost.

One of the main features of our scheduling approach is its interactive behaviour and its independence from the planning system. This approach is valid for every planning system, both forward and backward chaining planners. The scheduler allows

the integrated system to prune partial plans, avoiding the generation of invalid plans. Furthermore, the scheduler is able to manage several partial plans, which have been generated by the planner. Another important feature is the use of heuristics, characteristics of the scheduler, which will speed up the integrated process and improve its behaviour.

In this section we have presented the introduction to this paper. We propose the integrated system, its main features such as problem specification language (to model the problems), and its architecture in section 2. The scheduling process, its behaviour through an example and the way we manage all the temporal constraints and resource availability are described in section 3. Conclusions are discussed in section 4.

2 THE INTEGRATED SYSTEM

In this section we expose a high-level general view of our integrated system (Figure 1). The domain representation is obtained from the problem domain by means of the specification language. The domain representation consists of the problem objects (including the resources), the actions and the problem constraints. The integrated system of planning and scheduling solves the problem in order to achieve the executable plan. Nevertheless, new problem constraints or incidences may appear during the execution period. In this case, a reactivity stage is needed to obtain a new optimal plan according to the new problem constraints. All these elements are detailed below.

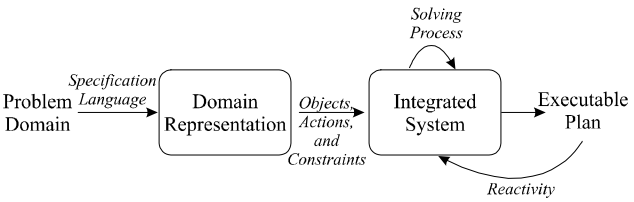


Figure 1. General view of the integrated system (from [13])

2.1 Problem Specification Language

In order to model and analyse the problem domain, we use a language, which allows the user to define the next elements:

- Domain object hierarchy. Classic approaches in planning use a declarative-language by means of first-order predicates for domain description [19]. In contrast to these schemes, we always maintain the same structure for literals [13]:

  <*class-name*> <*object*> <*slot-name*> <*value*>

  This frame-based structure allows us to model real application environments. The object hierarchy can represent problem objects as well as the resource hierarchy. There exists a special class for shared resources as in [23]. The resources are shared albeit nonsimultaneously by the actions. If objects are resources there are several slots by default, such as *quantity* (number of items), *resource availability* (temporal constraints that indicate when the resource can be used), *service* time (how long the resource is used by default), etc. Moreover, the user can also define the initial situation and goals to achieve by using the previous structure.
- Actions. Actions can be primitive actions, which cannot be divided any further or macro-actions, which group primitive actions in an established partial or total order. We can refine every macro-action in its primitive actions carrying out the planning and scheduling process through a hierarchy of different levels. Thus, we can obtain an initial plan that will be detailed in following steps of the process by means of a refinement method [8].
- Problem constraints. These constraints can be applied to different elements of the problem:
  - Temporal constraints over the entire plan. They indicate the possible execution duration of the plan by means of its possible beginning and ending.
  - Constraints over the resource usage. Due to the fact that resources cannot be simultaneously used by more than one action, constraints must guarantee that actions do not use the same resource in the same time.
  - Constraints over the ordering of actions because there are actions that must be executed in a specific order.
  - Constraints over the action durations. These durations can be dependent of the order in which they are executed.
  - Temporal constraints over the problem objects and their states (attributes). For instance, an object cannot be held in the same state for more/less than a determined interval, an object must reach a specific state in a determined moment, etc.

2.2 Architecture of the Integrated System

The planning and scheduling modules in the integrated system (Figure 2) share data structures of the system with the common information. This shared common information is stored in a special kind of data structure similar to a blackboard model [17]. The planner must accede to each data related to actions, their ordering, initial situation and goals. On the other hand, the scheduler must keep all the information related to allocation and nonsimultaneous resource usage, temporal constraints and ordering among actions.

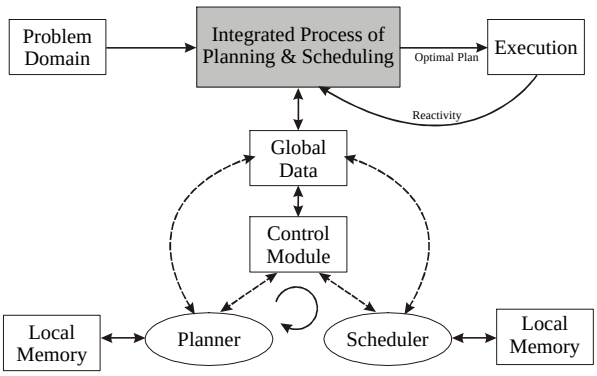


Figure 2. Integrated architecture of planning and scheduling

In the integrated system, a strong communication should be carried out during the construction process of each partial plan. The planning and scheduling processes exchange information, by means of the shared data structures, in order to obtain a more efficient integration. Communication between the planner and the scheduler must occur:

- Every time a new planned action (or an existing one) solves a precondition. In this case, the scheduler must update the ordering among the actions (and the ordering of the resources which are used by these actions) according to the new causal-link.
- When the planner demands a resource that must be used in an action. Here, the scheduler must update the sequence of utilisation of the resources in order to avoid a simultaneous usage.
- When a new ordering among actions is established due to a planning conflict resolution. As in the first case, the scheduler must update the ordering among the actions.

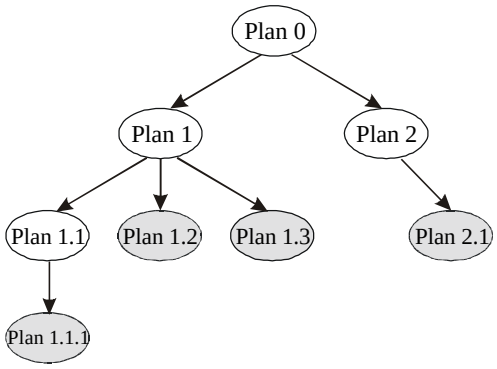


Figure 3. Search among alternative plans (current feasible partial plans are shaded)

In all the previous cases, the planner must inform the scheduler about:

- The action to be planned. Since the planner works on various excluding alternative plans (Figure 3), the planner must specify in which alternative plan the action is used. Each alternative plan represents a plan with different actions to achieve the problem objective. Only resources in actions of a same partial plan must not be simultaneously used. Therefore, the scheduler must be able to manage the possible excluding alternative plans at the same time, which are shaded in Figure 3.
- The resource list to be used in this action. The planner may know the resource list to use due to another previously planned action in the same partial plan or because the planner requires some specific resources. In this case, the scheduler must not allocate new resources, but it will check if this allocation is consistent with the known constraints for the required resource in its partial plan. If the planner does not know the resource to be used, the scheduler should assign the necessary resources for the action, in accordance with the action specification, by using optimization criteria.
- The order of the action to be planned. The planner may establish an order, both partial and total, among several actions of each partial plan. This order is related to other actions involved in its plan. This mechanism permits us to establish some ordering criteria by indicating that one action precedes another. If this ordering is the result of solving an ordering conflict, the scheduler must reorder the actions and determine if the new ordering is consistent according to the existing constraints. For instance, if an ordering is not feasible because of the violation of any temporal constraint or because there is not any available resource, this plan will be discarded.

When the scheduler detects an inconsistency, it communicates to the planner to discard this action. Furthermore, the scheduler recommends the planner a list of alternative available resources. In addition, the scheduler could suggest which partial plan (from the frontier of the plan tree (**Figure 3**)) should be expanded, taking into account which partial plan is less constrained or imposes less constraints over resources. This feature can be a valuable heuristic for improving the planning process. Hence, a great level of integration is required between the planning and scheduling processes. For instance, let be P_0 the current partial plan obtained as a result of a search in the space of partial plans (Figure 4). This plan is expanded by adding new actions or steps that will achieve the final objective. Several alternative plans P_a, P_b and P_c might be generated, which introduce new steps S_a, S_b and S_c, respectively. If the plan P_b is selected, the scheduler must check that its temporal constraints are satisfied and allocate the resources (only if it is necessary). In order to guarantee the constraints of a plan, the scheduler may constrain the partial sequence of steps of that plan. Therefore, the plan P_b' is the plan P_b with all its constraints satisfied. Next, the plan P_b' might be expanded, the plans P_{b1}', P_{b2}' and P_{b3}' would be generated and the process would continue until accomplishing all the problem objectives. Finally, the integrated system obtains an executable (and eventually optimal) plan according to the resource usage and other optimization criteria.

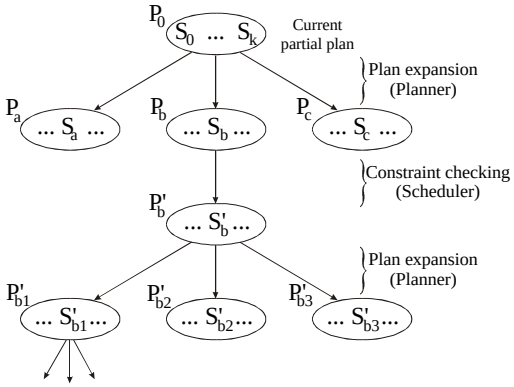


Figure 4. Process of plan expansion and constraint checking by the planner and the scheduler, respectively

During the execution time, some incidences might appear (for instance if some resource becomes unavailable). In this case, a new reactivity process to repair all the conflicts would be necessary to obtain a reassignment of resources, if possible.

Otherwise, the plan should be modified to accomplish the new problem requirements. This gives rise to a rescheduling or replanning problem, which is solved by means of a repair method [8].

As a result of the integration, we can obtain the following advantages:

- We can detect an inconsistency (due to resource unavailability, temporal constraint violation, etc.) in a partial plan as soon as this inconsistency appears. Since this inconsistency implies a nonfeasible partial plan, this plan is quickly discarded. Hence, the efficiency of the global process is improved.
- Since the system uses a specific module of scheduling, the system can manage more complex constraints than temporal planners. Moreover, we can define optimisation criteria (both in the planner and the scheduler) which improve both the efficiency of obtaining the plan and the quality (optimality) of the obtained plan.
- According to the way of obtaining the partial plans (Figure 4), the final plan is executable.

3 THE SCHEDULING PROCESS

Once established this general architecture, we will analyse the requirements that the scheduling process requires. On one hand, one of the main aims of the scheduling process is to guarantee the executability of the achieved plan according to available resources, context constraints, etc. [10]. All temporal constraints in the problem must be satisfied. Furthermore, the shared resource usage must also be consistent; i.e. enough available resources must exist to ensure the fulfilment of each planned action when it is finally executed. On the other hand, another aim is to guarantee the optimality of the obtained plan, according to its cost, due times and some optimization criteria.

In order to carry out the integration defined in the previous section we need a scheduler with a special behaviour, which should be more dynamic and interactive than traditional scheduling processes. Traditional schedulers are based on *Constraint Satisfaction Problems* or *CSP*s [16, 20]. These schedulers are not directly applicable here, because of its lack of flexibility: it is very costly to have to obtain a new solution every time a new constraint is added or eliminated. For each new set of constraints (which are the result of including or excluding constraints), a CSP process must resolve the entire problem in order to obtain a new solution. However, when the set of constraints is modified, the previous solution may become invalid. It is clear that incremental CSP methods might be used here, but we do not need the solution to the problem in each step [26]. We only need to assume consistency at each new asserted constraint. Furthermore, the scheduler must be *contextual*, i.e., it must be able to manage several alternative plans with complex temporal constraints simultaneously.

Main features of our scheduler, its behaviour through an example and the way of managing the temporal constraints are detailed in this section.

3.1 General Considerations

Temporal constraints in our problem can be represented by a temporal network where nodes represent time points and arcs between them represent disjunctive metric-temporal constraints [9]. Working with disjunctive metric-temporal constraints is a complex task because of it implies working with a huge number of equivalent networks (one for each disjunction). For instance, in a typical problem of scheduling, if the number of tasks on a common shared resource is n, the number of equivalent different nondisjunctive graphs for each generated graph will be 2^n [21].

Our scheduler uses a module for reasoning with temporal constraints, the *Temporal Constraint Network Manager* (*TCNM*). The TCNM guarantees the consistency of the temporal network by using a closure process, which propagates each new constraint to all nodes of the network [2].

In traditional scheduling systems, the entire set of problem constraints is known in advance, so the aim of a scheduling process is to obtain a solution which satisfies these constraints. Here, *CSP* techniques are usually used [16, 20]. Alternatively, other schedulers work on a set of initial solutions which may not satisfy the problem constraints, and the schedulers repair them over time [8, 23].

In opposition to these traditional scheduling processes, the problem constraints are incrementally supplied, in our case, by the planner while each partial plan is being generated. At each new constraint, the scheduler guarantees the consistency of all the currently known constraints in each partial plan. Our scheduler works in a progressive way. As in [6], schedules are constructed by a process of iterative refinement. We believe this approach to be more flexible because it allows us to add constraints in a dynamic way. When a new constraint is added into the system, the scheduler will detail the schedule constraining the domain of possible values. The scheduler does not obtain the solution that satisfies all the current constraints after each constraint is asserted, but maintains all the minimal sets of values that might be solutions. An inconsistency is produced when the domain of possible values becomes empty by the effects of a constraint C_i. For instance, we will see an example of two actions which use the same nonshared resource in Figure 5.

Let A_x and A_y be two actions that can be executed in any order of precedence and that use the same resource. Therefore, A_x must be executed before A_y (a) or vice versa (b). At this moment, the scheduler only maintains the interval of possible solutions in the timeline but without allocating any concrete value as in traditional *CSP*s. Next, if a constraint indicates that A_x must be executed before A_y, the scheduler will discard the established order "A_y before A_x". With this information, if a new constraint asserts the fact "A_y before A_x" this constraint will be treated as an inconsistency and it will be discarded. Furthermore, the scheduler may impose more restrictive constraints: it may impose additional constraints on plans due to temporal or resource usage constraints.

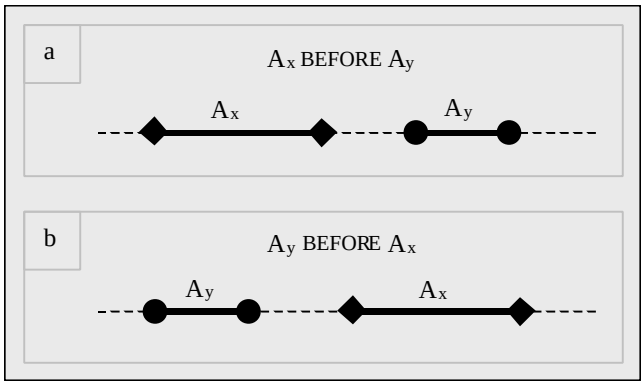


Figure 5. Example of two actions A_x and A_y which use a nonshared resource

3.2 Scheduler Behaviour

The scheduler must represent the necessary information in its temporal graph for each new planned action. Every action of each plan is translated into one or more temporal constraints, which are managed by the TCNM. The consistency can be guaranteed thanks to this TCNM which checks each new constraint introduced into the scheduling system.

Following, let us see the three levels of temporal constraints that arise in this approach.

Temporal constraints in the action level

At this level, constraints are based on durations of actions, ordering relations and constraints among their time points.

For every action (A_i), the scheduler must create two new time points, which represent the start time point ($A_i.on$) and the end time point ($A_i.off$) of the respective action. The disjunctive durations are represented by $A_i.on$ *{(dmin₁ dmax₁), (dmin₂ dmax₂), ..., (dminₙ dmaxₙ)}* $A_i.off$. Each interval represents a different disjunction in the duration of that action. Furthermore, the scheduler will check the ordering constraints with other actions. For instance, if there is another action A_j that must be executed before A_i, the constraint $A_j.off$ *{(0 ¥)}* $A_i.on$ will have to be satisfied. On the other hand, if an action A_j must be executed before a time point TP_j, it is represented by $A_j.off$ *{(0 ¥)}* TP_j.

Temporal constraints in the resource level

Here, constraints are based on durations and nonintersection of shared resources due to their nonsimultaneous usage. In addition, constraints among actions and time points of resource usage are included.

For every resource (R_x) used in an action (A_i), the scheduler must create two time points. They represent the start time point ($R_xA_i.on$) when the resource in that action may start to be used and the end time point ($R_xA_i.off$) when the resource in that action may have finished being used. The durations (resource usage) are represented by disjunctive intervals between the beginning and the ending of the resource usage: $R_xA_i.on$ *{(dmin₁ dmax₁), (dmin₂ dmax₂), ..., (dminₙ dmaxₙ)}* $R_xA_i.off$. The beginning and the ending of the resource utilisation is related to the start and end time points of the action they belong to. Hence, a resource may be used during all the action or only during a part of it. There may be an offset, either positive or negative, between the start point (end point) of the action and the start point (end point) of the resource in that action. Besides, it is necessary to guarantee that the use of a resource R_x in an action A_i (represented by R_xA_i) is not simultaneous with the use of the same resource in another action A_j (represented by R_xA_j). It can be performed simply by indicating the usage of the resource R_x in A_i is before or after the usage of that resource in A_j.

Temporal constraints in the attribute level

At this level, the constraints between the object's states (attributes) are represented. Constraints appear due to relations between states of one or more dynamic attributes. For instance, an attribute cannot change its value during a time window.

For every pair state-attribute (St_kAtt_x), the scheduler will create two new time points, which represent the beginning ($St_kAtt_x.on$) and the ending ($St_kAtt_x.off$) of an attribute state. In a similar way, the duration of these attribute states can be represented as $St_kAtt_x.on$ *{(dmin₁ dmax₁), (dmin₂ dmax₂), ..., (dminₙ dmaxₙ)}* $St_kAtt_x.off$. If there are some ordering relations among states, we use the type of constraints $St_kAtt_x.off$ *{(0 ¥)}* $St_lAtt_x.on$ if the attribute is the same. If the attributes are different, we use the constraint $St_kAtt_x.off$ *{(0 ¥)}* $St_lAtt_y.on$.

Since these changes in attribute states are produced by the effects of some actions, each pair state-attribute must be related to the action that produced its change. As in the resource level, there may be a delay, either positive or negative, between the beginning or ending of the action, and the change of the attribute state. The notion of persistence can be represented by a delay (see Figure 6) in the ending of the state change (if the persistence is positive the value $St_kAtt_x.off$ will be later than the ending of the action, which produces the change in this attribute state).

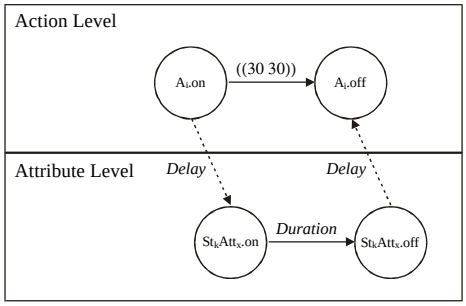


Figure 6. Representing the persistence in the attribute level

As we can observe from above, the three levels are quite similar. In fact, the steps to carry out these levels are practically the same: creating the time points, establishing the duration and managing the precedence relations.

Moreover, it is important to realise these three levels are repeated in each alternative plan managed by both the planner and the scheduler. The time points of different plans must not be related because they represent distinct alternatives to achieve the final plan. If some inconsistency is produced in a determined

plan, the planner will discard that plan and the scheduler will discard the graph with the three levels of that plan.

3.3 Illustration Through a Simple Example

In order to illustrate better the previous levels of temporal constraint management, we show a simple example of how the actions planned by the planner are managed by the scheduler.

```
(defclass FERRY (subclass-of RESOURCE)
   (slot location (type PLACE))          ;where is the ferry
   (slot status (type string))           ;is it empty
   (slot load-time  (type disjunctive-interval))
   (slot sailing-time  (type disjunctive-interval))
   (slot unload-time  (type disjunctive-interval)))

(defclass VEHICLE ()
   (slot location (type PLACE))          ;where is the vehicle
   (slot on-ferry (type FERRY))          ;in what ferry
   (slot on-lorry (type LORRY)))         ;in what lorry

(defaction sail (?n-ferry ?place1 ?place2)
   (vars (FERRY ?n-ferry sailing-time ?st))
   (preconds (FERRY ?n-ferry location ?place1))
   (test (!= ?place1 ?place2))
   (effects
        (add (FERRY ?n-ferry location ?place2))
        (delete (FERRY ?n-ferry location ?place1)))
   (duration ?st)
   (resources (ferry ?n-ferry 'during ?st)
              (bridge B1 'during ?st)))  ;B1 is the bridge

(defaction unload (?n-ferry ?n-vehicle ?place)
   (vars (FERRY ?n-ferry unload-time ?ut))
   (preconds (VEHICLE ?n-vehicle on-ferry ?n-ferry)
             (FERRY ?n-ferry location ?place))
   (effects
        (add (VEHICLE ?n-vehicle location ?place)
             (FERRY ?n-ferry status 'empty))
        (delete (VEHICLE ?n-vehicle on-ferry ?n-ferry)))
   (duration ?ut)
   (resources (ferry ?n-ferry 'during ?ut)))
```

Figure 7. Fragment of the problem specification: generic resource 'ferry', 'vehicle' and actions 'sail' and 'unload'

We will use as example a modified version of the well-known ferry problem [3]. The objective is to transport some vehicles from the margin of a river to the other one. In our case, we also have some lorries to carry out this task. Therefore, the resources are the ferries, the lorries and a bridge. This bridge must be up in order to the ferries can sail under it and it must be down in order to the lorries can drive along it. Thus, the bridge is a nonshared resource *used* by both the ferries and the lorries.

The user can define the actions, resources and actions of this problem by using the specification language defined in section 2.1. For instance, in Figure 7 appears the definition of some objects of the problem, such as the generic resource 'ferry', 'vehicle' and the actions 'sail' and 'unload. The action 'sail' transports a 'vehicle' from the margin of the river to the other one in a 'ferry' and the action 'unload' disembarks a 'vehicle' from the 'ferry'. In this example, there are not any constraints over the attributes. Even though the action durations might be disjunctive intervals, we do not use them in order to simplify the figures.

Let us suppose that the first action the planner plans is 'unload' which uses the ferry F1. Here, the temporal network that the scheduler manages is shown in Figure 8.

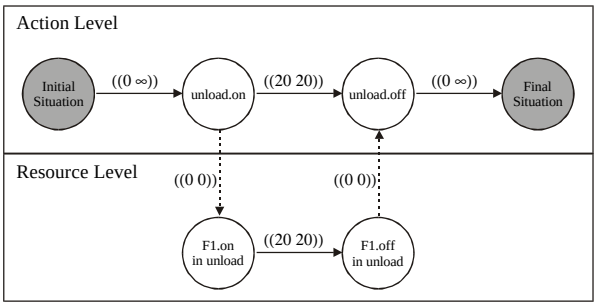


Figure 8. Action and resource levels of the action 'unload' managed by the scheduler

Next, if the planner plans (in the same alternative plan) the action 'sail' using the same ferry F1, the new constraints asserted in the temporal network are what appear in Figure 9. In this figure, we can see the temporal constraint *((-¥ -21) (16 ¥))* which implies using the ferry F1 in a nonsimultaneous way. For this reason, F1 will be used in the action 'unload' either 16 units of time after its use in 'sail' or 21 units of time before, but no simultaneously.

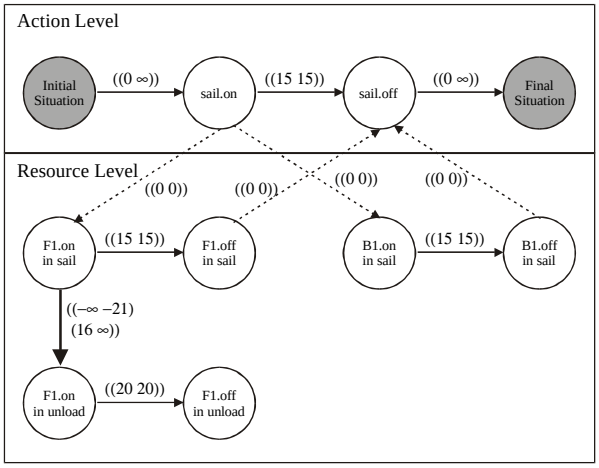


Figure 9. Action and resource levels of the action 'sail' managed by the scheduler

3.4 Management of Temporal Constraints

The temporal constraints over actions, resources and attributes are treated as disjunctive metric-temporal constraints. Therefore, we have two possible alternatives to manage them and to guarantee their consistency and correctness. According to the

behaviour of these constraint management algorithms we can classify them into two different kinds: algorithms that maintain the derived constraints and algorithms that only maintain the input constraints [21].

Algorithms that maintain the derived constraints

These algorithms require large amounts of memory to store all the generated constraints in the closure process. The reason of maintaining the derived constraints is to allow us to know quickly (without any additional process) which is the temporal constraint between two time points. When a new constraint between two temporal points is asserted into the system, these algorithms check its consistency with the existing constraint. If the new constraint does not violate the existing one, the resulting constraint will be the more restrictive combination between them. There are several levels of consistency, typically path-consistency (which guarantees the consistency of all the paths between two temporal points) and global consistency [9] which guarantees the minimality of the network.

The propagation (deriving new constraints) is carried out by means of the closure process detailed in [2] and it is graphically represented in Figure 10. Briefly, each time a new constraint is asserted between the time points *i* and *j*, the following loops are executed:

- Loop 1. The derived constraint between node i and node kn is calculated:

$$C_{ik_n} = C_{ik_n} \oplus (C_{ij} \otimes C_{jk_n}),$$

  where the operation $\oplus$ is the intersection operation, and $\otimes$ is the combination operation [9].

- Loop 2. The derived constraint between node j and node lm is calculated:

$$C_{jl_m} = C_{jl_m} \oplus (C_{ji} \otimes C_{il_m})$$

- Loop 3. The derived constraint between node lm and node kn is calculated:

$$C_{l_m k_n} = C_{l_m k_n} \oplus (C_{l_m j} \otimes C_{jk_n})$$

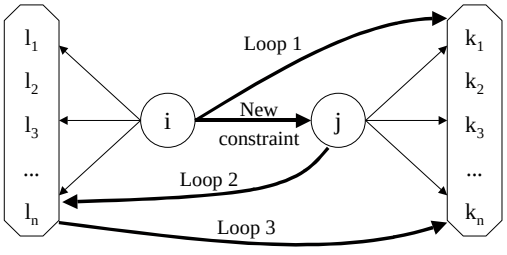


Figure 10. Closure process which propagates the effects of asserting a new constraint

Since guaranteeing the consistency in disjunctive networks is a very complex task (NP-complete complexity), we can decrease its complexity by using algorithms (of polynomial complexity) that relax the consistency. The former algorithms perform the propagation process in a faster way, but do not guarantee a consistent solution. Hence, we cannot assure each new constraint is inconsistent.

Algorithms that only maintain the input constraints

These algorithms may be used with the aim of reducing the complexity of adding new constraints. The main advantage of these algorithms is that they do not require large amounts of memory because they do not maintain the derived constraints. These algorithms do not carry out any propagation process among the new constraint and the existing ones in the network. Therefore, they only maintain the asserted constraints. This kind of algorithms eases the process of retracting some asserted constraints into the system. When a new constraint between two temporal points is inserted into the system, the algorithm retrieves the minimal constraint between these two primitives. Next, the algorithm checks if the new constraint is consistent with the retrieved one. If it is consistent, the new constraint is accepted, and if not it is rejected. The main problem of this approach is to calculate the minimal constraint in a nonpropagated network. It is a complex task because there exists an exponential number of paths that represent the constraints to find. Currently, we are working on new algorithms to calculate this minimal constraint in a disjunctive network in an efficient way on the basis of [21].

4 CONCLUSION THROUGH RELATED WORK

The main drawback of temporal planners for performing scheduling tasks is that handling difficult temporal constraints over plans, actions and shared resources becomes in a complex task because they do not have specific temporal managers. Usually, they use a kind of *Time Point Network* [5] to represent time constraints on time points. Although some temporal planners can work with metric constraints, such as IxTeT [12], traditionally they do not have the enough temporal knowledge or they do not use disjunctive constraints. In opposition, if specific processes of planning and scheduling are used, we will be able to apply characteristic features of each process. For instance, in the planner, techniques to diminish the search space and in the scheduler, more efficient criteria to carry out a better optimisation of the obtained schedule.

On the other hand, CSP and incremental CSP techniques [26] are not applicable enough due to the fact that they obtain a new solution instead of guaranteeing only the consistency of the problem constraints. Other authors have modelled plans as a set of constraints (future and plan entity constraints) which together limit the behaviour of the plan during its execution [25].

As we can see, much effort has been performed in order to manage resource scheduling in an efficient way [24]. Moreover, many attempts of integrating planning and scheduling have been carried out, mainly in the works of Muscettola and Smith with *HSTS* [18, 22] and Gervasio [14]. According to [8], we propose an integrated refinement system similar to the one proposed for Ghallab in [12]. Our system works with different alternative plans (*contextual scheduling*) and with disjunctive constraints to schedule resources avoiding their simultaneous usage. Hence, in this paper we have detailed the behaviour of our scheduler in our planning and scheduling integrated environment. The scheduler must have a dynamic and interactive behaviour different from

traditional *CSP*s. Because the planner in a planning environment frequently provides the scheduler new constraints, the scheduler must validate them taking into account the plan they belong to. We have presented an easy way to manage all the constraints of the problem, both temporal constraints and resource usage constraints. They are managed by three very similar levels: action level, resource level and attribute level. In addition, we have discussed two ways to manage the temporal constraints: maintaining the derived constraints and maintaining only the input ones. Currently, we are working on nonpropagation techniques which allow us to retract constraints in a very efficient way. We are also studying the possibility of adding more expressive temporal constraints to the scheduler [15].

ACKNOWLEDGEMENTS

This work is proposed in the *Intelligent Planning & Scheduling Group* of the Polytechnic University of Valencia (http://www.dsic.upv.es/users/ia/gps) and partially supported by the grant CICYT/TAP98-0345 from the Spanish government.

REFERENCES

[1] C. Bäckström, 'Computational Aspects of Reordering Plans', *Journal of Artificial Intelligence Research*, **9**, 99-137, (1998).

[2] F. Barber, 'Reasoning on complex disjunctive temporal constraints', *Journal of Artificial Intelligence Research*, **12**, 35-86, (2000).

[3] A. Barret, D. Christianson, M. Friedman, K. Golden, S. Penberthy, Y. Sun and D. Weld, *UCPOP v4.0 user's manual*, Technical Report TR 93-09-06d, Dept. of Computer Science and Engineering, University of Washington, Seattle, WA, (1996).

[4] H. Beck, *TOSCA: A novel approach to the management of job-shop scheduling constraints*, Realising CIM's Industrial Potential: Proceedings of the Ninth CIM-Europe Annual Conference, 138-149, (1993).

[5] H. Beck and A. Tate, *Open Planning, Scheduling and Constraint Management Architectures*, The British Telecommunication's Technical Journal, Special Issue on Resource Management, (1995).

[6] M. Boddy, 'Temporal Reasoning for Planning and Scheduling', *SIGART-ACM Bulletin*, **4(3)**, 17-20, (1993).

[7] K. Currie and A. Tate, 'O-Plan: The open planning architecture', *Artificial Intelligence*, **52(1)**, 49-86, (1991).

[8] T.L. Dean, L. Greenwald and L.P. Kaelbling, *Time-Critical Planning and Scheduling Research at Brown University*, Technical Report CS 94-41, Dept. of Computer Science, Brown University, (1994).

[9] R. Dechter, I. Meiri and J. Pearl, Temporal constraint networks, *Artificial Intelligence*, **49**, 61-95, (1991).

[10] J. Dorn and K. Froeschl, eds. *Scheduling of Production Processes*, Ellis Horwood, (1993).

[11] A. El-Kholy and B. Richards, *Temporal and Resource Reasoning in Planning: the parcPLAN approach*, ECAI 96, 12th European Conference on Artificial Intelligence, 614-618, (1996).

[12] M. Ghallab and H. Laruelle, *Representation and control in IxTeT, a temporal planner*, In Hammond, 61-67, (1994).

[13] A. Garrido, E. Marzal, L. Sebastiá L. and F. Barber, *Un Modelo de Integración de Planificación y Scheduling*, In Proceedings of CAEPIA'99 1(3), 1-9, (1999).

[14] M. Gervasio and G. DeJong, *A Completable Approach to Integrating Planning and Scheduling*, Artificial Intelligence Planning Systems: Proceedings of the First International Conference (AIPS 92), pp. 275-276, Morgan, Kaufmann, (1992).

[15] P. Jonsson and C. Bäckström, 'A unifying approach to temporal constraint reasoning', *Artificial Intelligence*, **102**, 143-155, (1998).

[16] V. Kumar, 'Algorithms for Constraint Satisfaction Problems: A Survey', *AI Magazine*, **13(1)**, 32-44, (1992).

[17] T.J. Laliberty, D.W. Hildum, N.M. Sadeh, J. McA'Nutty, D. Kjenstad and S.F. Smith, *A Blackboard Architecture for Integrated Process Planning and Production Scheduling*, In Proceedings of ASME Design for Manufacturing Conference, Irvine, CA, (1996).

[18] N. Muscettola, *HSTS: Integrating Planning and Scheduling*, Morgan Kaufmann, San Mateo, CA, (1994).

[19] S. Penberthy and D.S. Weld, *UCPOP: A sound, complete, partial-order planner for ADL*, In Proceedings of the 1992 International Conference on Principles of Knowledge Representation and Reasoning, 103-114, Kaufmann, Los Altos, CA, (1992).

[20] N.M. Sadeh and M.S. Fox, 'Variable and value ordering heuristics for the job shop scheduling constraint satisfaction problem', *Artificial Intelligence*, **86**, 1-41, (1996).

[21] M.A. Salido, A. Garrido and F. Barber, *Evaluation of Algorithms to Satisfy Disjunctive Temporal Constraints in Planning and Scheduling Problems*, In Proceedings of AISB'00 Symposium on AI Planning and Intelligent Agents, (2000).

[22] S. Smith, *Integrating Planning and Scheduling: Towards Effective Coordination in Complex, Resource-Constrained Domains*, Italian Planning Workshop, Rome, Italy, (1993).

[23] S.F. Smith, O. Lassila and M. Becker, *Configurable, Mixed-Initiative Systems for Planning and Scheduling*, In Advanced Planning Technology, Menlo Park, AAAI Press, (1996).

[24] B. Srivastava and S. Kambhampati, *Efficient Planning Through Separate Resource Scheduling*, AAAI Spring Symp. on Search Strategy under Uncertain and Incomplete Information, (1999).

[25] A. Tate, *Representing Plans as a Set of Constraints –the <I-N-OVA> Model*, In Drabble, B., ed., Proc. Third Conference on Artificial Intelligence Planning Systems (AIPS 96), 221-228, (1996).

[26] E. Tsang, *Foundations of constraint satisfaction*, Academic Press, (1993).

[27] G. Verfaillie, S. de Givry and D. Lesaint, *What is New in On-Line Scheduling?*, In On-line scheduling characteristics (Working document), available in http://www.lcr.thomson-csf.fr/projects/planet/ols-tcu.html, (1999).

Heuristic Methods for Solving *Job-Shop* Scheduling Problems

A. Garrido, M. A. Salido, F. Barber and **M. A. López**[1]

Abstract. Solving scheduling problems with Constraint Satisfaction Problems (CSP's) techniques implies a wide space search with a large number of variables, each one of them with a wide interpretation domain. This paper discusses the application of CSP heuristic techniques (based on the concept of slack of activities) for variable and value ordering on a special type of job-shop scheduling problems in which the operations must schedule inside of temporal windows. These techniques are improved by introducing the concepts of slack probability and the find-hole method. Thus, a more flexible heuristic technique is obtained, which improves empirical efficiency and allows early detection of unfeasible problems.

1 INTRODUCTION

Scheduling is the problem of allocating limited resources to operations (activities) over time. Scheduling is a complex task that can be formulated using a constraint-based representation. Reasons for scheduling complexity include [4]:

- Scheduling is a feasibility problem. The final solution must accomplish all the problem constraints. Another objective to be satisfied is the optimization of an evaluation function, adjusting to certain criteria as cost, lateness, process time, inventory time, etc.
- Some scheduling problems have many constraints due to the unavailability of resources, due dates, etc.
- Constraint representation cannot express the importance of the value domains. The number and identity of tasks that require a resource over a particular time interval is a key piece of information that can suppose the basis for heuristic variable and value orderings.

Scheduling constraints are usually disjunctive ones (i.e.: two tasks cannot use the same resource at the same time). The consistence problem of metric disjunctive constraints is NP-hard [3], such that *CSP* techniques are used. However, inequality constraints generate a large search space that may have few (or no) feasible solutions. Thus, it becomes necessary to define techniques to empirically decrease this complexity and be able to solve real problems more efficiently (constraining value domains, relaxing some constraints, etc). We are interested in the relaxation of the heuristic variable and value orderings to obtain a more flexible method, which makes a better use of the knowledge the scheduler may have about each particular problem. Our work is focused on *job-shop scheduling* problems. In these problems, operations must be scheduled within their feasible time windows (i.e.: between its respective earliest start time and latest finish time). We propose heuristic techniques for variable and value orderings to be included in two known searching algorithms: *Basic-Depth-First Backtrack and Depth-First-with-DCE* [9, 10]. The former algorithm is the classical chronological backtracking procedure heuristically improved. The latter uses the additional heuristic *Dynamic Consistency Enforcement* (*DCE*), which dynamically focuses its effort on critical resource subproblems and learns from its previous faults.

We summarize the main concepts about the *job-shop scheduling* problem and the *CSP* approach in Section 2. In Section 3, we introduce the search method used and new heuristic concepts in this process. The proposed heuristics are empirically evaluated on a set of typical problems in Section 4. Conclusions and final remarks are discussed in Section 5.

2 THE *JOB-SHOP* SCHEDULING PROBLEM

A *job-shop scheduling* problem is represented by a set of jobs $J=\{J_1,...,J_n\}$ and a set of resources $R=\{R_1,...,R_m\}$. Each job J_i consists of a set of operations $O^i=\{O^i_1,...,O^i_n\}$ which must be performed between a *ready-time* (rt_i) and a *due-time* (dt_i). The execution of each operation (O^i_k) requires the use of a set of resources (R^i_k Í R) during a time interval (du^i_k). The start time st^i_k of operation O^i_k indicates when the operation may begin to use the resources R^i_k.

The problem of *job-shop scheduling* can be considered as a Constraint Satisfaction Problem [1], with the following elements:

- A set of variables $\{x_1,...,x_n\}$ associated with the start time of operations. These variables take values in finite domains $\{D_1,...,D_n\}$ that may be constrained by unary constraints over each variable. In these problems, time is usually assumed discrete, with a problem-dependent granularity.
- A set of constraints $\{c_1,...,c_m\}$ among variables which are predicates $c_k(x_i,...,x_j)$ defined on the Cartesian product D_i ´...´ D_j and restrict the variable domains.

[1] Dpto. Sistemas Informáticos y Computación, Universidad Politécnica de Valencia, Camino de Vera s/n 46071, Spain, email: {agarridot, msalido, fbarber}@dsic.upv.es

Constraints of job-shop scheduling problems can be represented as binary, disjunctive, metric and point-based constraints [3]:

$$c_i(x_j, x_k) \equiv \{[d_1^- d_1^+][d_2^- d_2^+] \cdots [d_n^- d_n^+]\} \text{ where } d_i^- \le d_i^+ .$$

This constraint disjunctively restricts the temporal distance between x_j and x_k:

$$d_1^- \le x_k - x_j \le d_1^+ \vee \cdots \vee d_n^- \le x_k - x_j \le d_n^+$$

Moreover, unary constraints on a variable x_j may be represented as binary constraints between the variable and a special time-point $T0$, which represents *'the beginning of the world'* (usually, $T0 = 0$):

$$x_j - T0 \in \{[d_1^- d_1^+][d_2^- d_2^+] \cdots [d_n^- d_n^+]\}$$

meaning that: $x_j \in \{[d_1^- d_1^+][d_2^- d_2^+] \cdots [d_n^- d_n^+]\}$

Thus, we have a Temporal Constraint Problem or *TCP* [3]. An assignment of the variables $(x_i,...,x_j)$ in their domains is *consistent* with respect to c_k iff c_k is satisfied. A solution of a *CSP* is an assignment of a value to each variable within its respective value domain satisfying all constraints.

2.1 Constraints and *CSP* Algorithms

Two main constraints appear in this kind of *job-shop* problems:

- *Precedence constraints*: the operations O_j^i of each job J_i must be scheduled according to precedence constraints, i.e., there exists a partial ordering among the operations of each job and may be represented (Figure 1) by a precedence graph or *tree-like* structure [10]. Each precedence constraint O^i_k *BEFORE* O^i_l gives rise to the linear inequality $st^i_k + du^i_k < st^i_l$.
- *Capacity constraints:* resources cannot be used simultaneously by more than one operation. Thus, two different operations O^i_k and O^j_l cannot overlap unless they use different resources. Capacity constraints give rise to disjunctive linear inequalities:

$$\forall\, O^i_k, O^j_l: \quad R^i_k \cap R^j_l = \emptyset \cup ([O^i_k \text{ before } O^j_l] \cup [O^j_l \text{ before } O^i_k]$$

Additionally, other technological constraints may restrict the set of possible execution times for individual operations or availability times for resources, etc.

There are two main objectives in a *job-shop CSP*:

- *Feasibility*: if a solution can be found. We can later be interested in optimal solutions, according to several scheduling optimality criteria or cost functions to evaluate the optimality of each feasible solution.
- *Efficiency of the CSP process*. This objective implies minimizing backtracking processes and, consequently, the conflicts. When an operation requires a resource that is already being used, a conflict appears. We can anticipate possible conflicts and improve the algorithm efficiency by detecting resources with the highest contention.

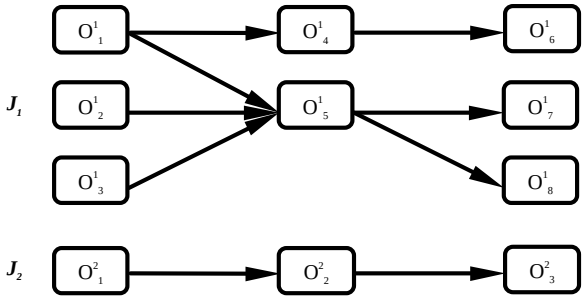


Figure 1. Example of tree-like (from [10])

The solving algorithm should be able to generate and evaluate all the possible assignments for each one of the problem variables. However, only a small fraction of these possible assignments will participate in a feasible solution. Thus, an efficient search, by means of an incremental method, should establish an appropriate order for instantiating the variables and obtain the order to select the values from the domains. This incremental method can be applied with two main techniques: retrospective and prospective techniques [1].

- *Retrospective techniques* that assign a value to a variable checking other variables with assigned values in order to avoid conflicts. If all constraints hold, another variable and value are selected. Otherwise, some constraint is not satisfied and backtracking occurs. The simplest retrospective technique is chronological backtracking. If the verification of consistency fails, it will select the variable most recently instantiated and it will test another value of its domain. When all values in the variable domain have been unsuccessfully tested, the backtracking goes back to the next most recently instantiated variable, and so on. If the procedure goes back to the initial state (i.e., the state with an empty schedule), the problem is unfeasible. Chronological backtracking may behave in an inefficient way, which is known as *thrashing* [6]. Thrashing may appear when the backtracking tries to recover from a *dead-end* state (a partial solution which cannot be completed). Since it tries to schedule the last scheduled operation, it may go back to similar *dead-end* states. However, the operation most recently scheduled is not usually the cause of the conflict. The search may fail due to some variable assignment previously performed in the search.
- *Prospective techniques* that propagate the effects of each variable instantiation to unassigned variables. This propagation is based on three levels of local consistency[2] which arise from the analysis of reasons for thrashing [1]:
  i) *Lack of node consistency*. No elements in the variable domain satisfy unary constraints. Assigning these values causes immediate failures.

[2] Local levels of *k-consistency* are empirically more efficient in backtracking processes than *total-consistency* [6].

ii) *Lack of arc consistency* which applies the previous concept to binary constraints.

iii) *Lack of path consistency*. For each value $x_i \hat{I} D_i$ and $x_k \hat{I} D_k$ such that $(x_i=v_i\ c_{ik}\ x_k=v_k)$ holds, a sequence of values does not exist

$x_{i+1} \hat{I} D_{i+1}, x_{i+2} \hat{I} D_{i+2},...,x_{k-1} \hat{I} D_{k-1}$, such as

$(x_i\ c_{i\,i+1}\ x_{i+1})$, $(x_{i+1}\ c_{i+1\,i+2}\ x_{i+2})$,..., and $(x_{k-1}\ c_{k-1\,k}\ x_k)$ hold.

2.2 Variable and Value Orderings

The order in which variables and domain values are selected in a *CSP* process is important to decrease the empirical computational time. An optimal variable/value ordering would produce a linear time solution for a feasible scheduling problem because no backtracking would be necessary. Thus, an aim is to minimize backtracking stages and, consequently, the conflicts. These conflicts appear when an operation requires a resource that is already being used. Therefore, it is necessary to use good *ordering* heuristics to efficiently solve practical problems and reduce the effective size of the search space [6]. In particular, *texture measurements* [8] can be used as a basis for heuristic decisions. A texture measure is an assessment of properties of a constraint graph and reflects the intrinsic structure of a particular problem. On the other hand, by detecting resources that have the highest contention, we can anticipate possible conflicts and improve algorithm efficiency [2, 10]. The most common heuristic is to instantiate the *most constrained* variable to *its least constraining* value. Intuitively, the earlier the most constrained variable is instantiated, the earlier the backtracking will take place (pruning the search space and minimizing thrashing). Furthermore, the probability of finding a solution without backtracking in this variable is increased by assigning the least constrained value.

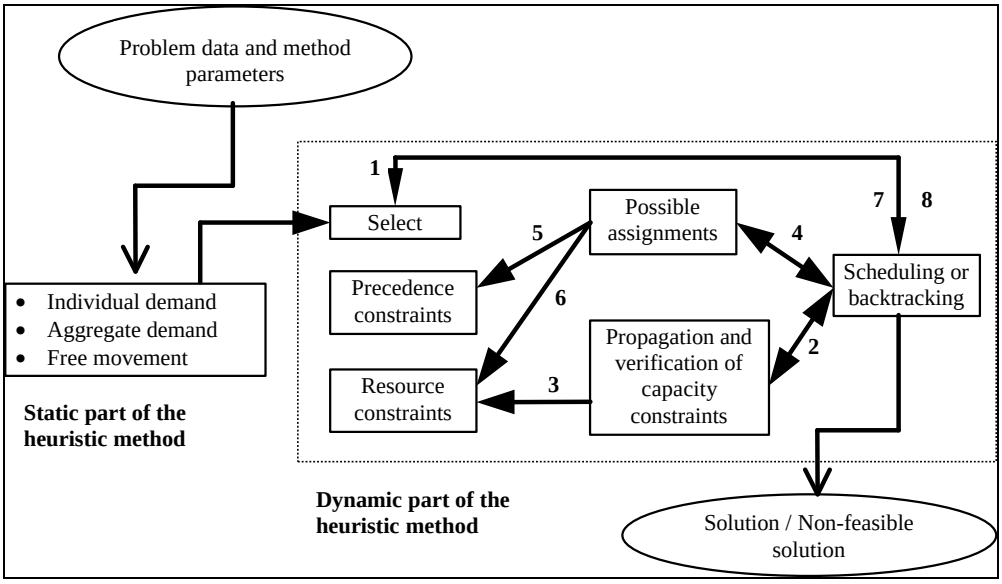


Figure 2. Heuristic Searching Method: Static part and Dynamic part

3 HEURISTIC SEARCH METHOD

We use the classic backtracking procedure, which is improved with specific heuristic criteria for variable ordering to avoid thrashing and to improve the efficiency of the process. Our work is based on the heuristic techniques developed by Sadeh [10]. However, we do not consider all the criteria because they vastly limit the search space. Moreover, the flexibility of the method is based on some additional considerations that allow us to extend the search space.

In order to study the calendar of utilization of each resource, we use the *Individual Demand* and *Aggregate Demand* techniques. These techniques and the concept of *slack probability* of an operation allow us to make better use of the temporal windows of the operations for scheduling. These techniques select the most critical operation and assign it its less-constrained value. We have mixed retrospective and prospective techniques in the verification of the consistency. Furthermore, we introduce the *find-hole* method that allows us to eliminate any doubt in the presence of a conflicting situation either: (i) according to operations not yet scheduled (prospective); or (ii) according to the maintenance of the consistency among the new operation and the operations already scheduled (retrospective).

3.1 Searching Process

Chronological backtracking is based on the incremental search of a solution. This incremental search can be represented as a transition between states. It starts in an initial state without any scheduled action. It finishes in a final state when a solution is found (all the actions have been scheduled) or when there is no feasible solution. Our algorithm uses some heuristics to improve the efficiency of the classical backtracking techniques. Figure 2 shows a schematic representation of the heuristic method used in

the search of a solution. The method has two parts: one static part (search anticipation), and another dynamic part, which is based on the techniques that are applied during the search process. The majority of the decisions are taken during the search process: consistency enforcing, which value to select, schedule, or unschedule, etc. However, all these decisions are conditioned by the decision adopted in the static part: the order of selecting the variables.

We can observe a possible sequence of decisions in the scheduling of an operation in Figure 2:

1. An operation is selected to be scheduled in (1).
2. Capacity constraints are verified according to the operations not yet scheduled and no conflict is detected (2, 3).
3. We look for a possible time for the start of the operation (4) guaranteeing the precedence and capacity constraints according to the operations that have already been scheduled (5, 6).
4. There is no conflict and the operation is scheduled (7).

On the other hand, if a conflicting situation is detected in step 3, a backtracking stage (8) will occur in step 4 and the procedure will go back to the most recently scheduled operation testing a new value of its domain.

3.2 *Slack* Probability of an Operation

The slack of an operation O^l_i is defined in Operational Research as the difference between the latest (lst^l_i) and earliest (est^l_i) start time:

$$S(O^l_i)=lst^l_i - est^l_i$$

This value indicates the number of units of time the operation's execution can be delayed, without delaying the project. In our context, we have a similar underlying idea: we are interested in finding which is the operation that admits more starting times, taking into account the precedence constraints. We define the *slack probability* (*SP*) of an operation as the probability of movement in the temporal window $[est^l_i,...,lft^l_i]$, where est^l_i is the earliest O^l_i start time and lft^l_i is the latest O^l_i finish time. Thus, the *slack probability* is defined as:

$$SP\left(O^l_i\right)=1-\frac{du^l_i}{\left(lft^l_i-est^l_i+1\right)}$$

If the operation has only one possible start time, the *slack probability* will be null. Otherwise, if the operation has a wide domain of possible start times, its *slack probability* will be the unit. Hence, an operation is more conflictive than another if its *slack probability* is minor.

A simple problem is shown in Figure 3. In this figure, there exist operations which share a same resource *R*. In addition, operation slacks are represented for each operation. If the criterion for selecting the next operation is the *slack probability*, the next operation to be scheduled will be the operation O^4_2 because its value $SP(O^4_2)$ is the least.

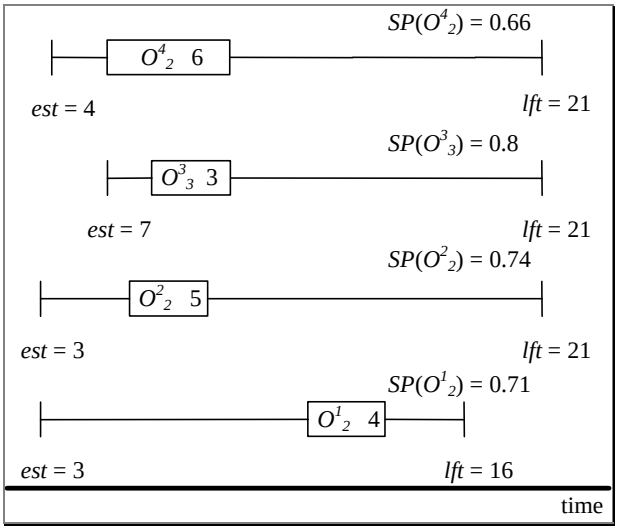


Figure 3. Representation of the slack probability for operations that share a resource *R* (boxes are labeled by the name of the operation and its duration)

3.3 Consistency Enforcing: the Heuristic *Find-Hole* Method

Once an operation has been selected to schedule, it is necessary to check that all the involved constraints are satisfied:

- *Consistency according to the precedence constraints*. The precedence constraints are defined among operations of a same job: the process must guarantee that two operations of a same job are not executed in the same instant of time. Essentially, the earliest start time is propagated downstream within the job whereas latest start time is propagated upstream. This propagation is applied before the *CSP* search process.
- *Consistency according to constraints about resource capacities*. Checking the consistency of capacity constraints is a difficult process due to the next constraints:
  - *Forward consistency checking*. When an operation is scheduled and a resource is allocated to the operation, a forward checking process analyzes the set of remaining possible reservations of other operations that requires the same resource and removes those conflicting reservations.
  - *Additional consistency check*ing. The process must check if two unscheduled operations that require the same resource are not overlapped. Two operations overlap when both require the same resource at the same time for every start time.
  - *Find-hole*. This method checks if an apparent conflicting situation is actually conflicting. Before indicating an operation has not any possible execution, it is necessary to check the resource usage calendar of this operation. If the shared resource is not used during the entire operation's execution, it can be used in another

operation. The *find-hole* method identifies the two extreme time points of the temporal line in which the action is executed (t^- and t^+, respectively) and searches some hole in which the resource is available and can be used by another operation. If a hole does not exist, an inconsistency will be detected because the resource is not available.

$$find\text{-}hole(O^l{}_i)\ checks\ if \begin{cases} lst^l{}_i + du^l{}_i > t^- \\ lst^l{}_i \le t^+ \\ est^l{}_i + du^l{}_i > t^- \end{cases}$$

4 EMPIRICAL EVALUATION

In this section, we study the empirical evaluation of the developed heuristic method. The empirical method performance is compared with the algorithms *Basic-Depth-First* and *Depth-First-with-DCE*. Both algorithms use the same variable/value ordering heuristics and the same techniques of consistency enforcing.

- *Basic-Depth-First*. This algorithm shows the efficiency of a chronological backtracking (it always goes back to the last scheduled operation).
- *Depth-First-with-DCE*. This algorithm behaves as the previous one, but it uses the added *DCE* heuristic.

4.1 Design of the Data Test

We have defined four types of *job-shop scheduling* problems. Each type has a different number of jobs, operations and resources (see Table 1). We have randomly generated 50 problems of each type. The number of operations may vary, but the number of jobs and resources cannot.

Table 1. Types of *job-shop* scheduling problems

	JOBS	RESOURCES	OPERATIONS
TYPE 1	10	6	60 max. 20 min.
TYPE 2	10	8	80 max. 50 min.
TYPE 3	10	10	100 max. 80 min.
TYPE 4	12	10	120 max. 108 min.

Two parameters allow us to deal with different scheduling conditions. The range parameter (*RG)* adjusts the distribution of the due dates and release dates of the jobs. The bottleneck parameter (*BK)* controls the number of bottleneck resources. In each problem type, we have three different values of the range parameter (*RG*) and two bottleneck configurations (*BK*) (see Table 2). Moreover, the operation durations were randomly obtained from two different distributions, depending on whether an operation requires a bottleneck resource or not.

Table 2. Comparison of the heuristic methods, which is used in the algorithms *Basic-Depth-First* and *Depth-First-with-DCE* vs. Chronological Backtracking (over 5 sets of 40 *job-shop* problems). Standard deviations appear in brackets

Performance of the heuristic method					
		CHRONOLOGICAL BACKTRACKING	**BASIC-DEPTH-FIRST**	**DEPTH-FIRST-WITH-DCE**	
				K=4	**K=8**
RG=0.2	**Search Efficiency (*)**	**0.12** (0.15)	**0.48** (0.44)	**0.68** (0.30)	**0.85** (0.33)
BK=1	**Solved Experiments**	**1**	**17**	**17**	**17**
RG=0.2	**Search Efficiency (*)**	**0.12** (0.15)	**0.64** (0.43)	**0.75** (0.32)	**0.86** (0.29)
BK=2	**Solved Experiments**	**1**	**24**	**24**	**24**
RG=0.1	**Search Efficiency (*)**	**0.17** (0.24)	**0.83** (0.33)	**0.89** (0.23)	**0.92** (0.19)
BK=1	**Solved Experiments**	**3**	**33**	**34**	**34**
RG=0.1	**Search Efficiency (*)**	**0.17** (0.24)	**0.91** (0.26)	**0.93** (0.20)	**0.94** (0.16)
BK=2	**Solved Experiments**	**3**	**36**	**36**	**36**
RG=0.0	**Search Efficiency (*)**	**0.15** (0.20)	**0.96** (0.20)	**0.96** (0.17)	**0.97** (0.14)
BK=1	**Solved Experiments**	**2**	**38**	**38**	**38**
TOTAL	**Search Efficiency (*)**	**0.15** (0.20)	**0.76** (0.34)	**0.84** (0.25)	**0.91** (0.23)
	Solved Experiments	**10**	**148**	**149**	**149**

(*) obtained by dividing the total number of problem's operations by the number of generated nodes in the solution search. In this way, the maximum efficiency is 1

4.2 Algorithm Comparison

Since a depth-bound was set in the solution search, when more than 800 states are generated, the search process stops. In this case, we assume the problem is probably nonfeasible.

The obtained results are summarized in Table 2. As we thought, the chronological backtracking method is not enough to solve complex *job-shop* problems. In spite of the low rate of found solutions, the efficiency is appropriate enough (it is 0.85 with *RG=0.2*, *BK=1* and consistency level *k=8*). This efficiency rate is due to the fact that we have also considered the unsatisfied problems in the efficiency calculation. Hence, the

proposed heuristic technique is capable of stopping the search when it deals with unfeasible problems (the process does not expect to find a feasible solution).

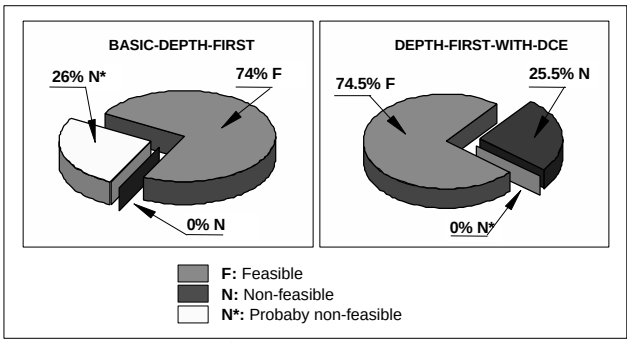


Figure 5. Comparison of the algorithms *BDF* and *DCE*

The results obtained by *Basic-Depth-First* (*BDF*) and *Depth-First-With-DCE* (*DCE*) algorithms are presented in Figure 4. 26% of the problems tested by *BDF* were declared probably nonfeasible (more than 800 states were generated), whereas *DCE* classified them as feasible and nonfeasible problems. Consequently, the *BDF* algorithm is not capable of properly determining whether the problem is feasible or not. However, the results are inverted in the *DCE* algorithm: all the problems for which the solution is not found are unfeasible.

5 CONCLUSIONS

In this paper, a variation of the *job-shop scheduling* problem, in which operations have to be performed within fixed temporal windows, has been analyzed. We refer to these problems as *job-shop CSP*s because operations must accomplish precedence and capacity constraints. *Job-shop CSP* problems cannot be solved with traditional scheduling techniques such as priority dispatch rules, one-pass scheduling techniques [5, 7] or traditional linear programming techniques of Operational Research.

Our studies are mainly based on Sadeh's work [8, 9, 10]. In his work, the *CSP* paradigm is applied to this kind of problems, demonstrating that *CSP* methods are promising alternatives to traditional scheduling methods for solving *job-shop CSP*s. We have proved propagation techniques and heuristics for variable and value ordering (*slack probability* and *find-hole* methods) are useful in solving *job-shop scheduling* problems. Moreover, these techniques are able to stop the solving process when a feasible solution cannot be found. Furthermore, the empirical results show these heuristic methods can efficiently solve different types of problems that could not be solved with the traditional *CSP*s.

ACKNOWLEDGMENTS

This work is proposed in the *Intelligent Planning & Scheduling Group* of the Polytechnic University of Valencia (http://www.dsic.upv.es/users/ia/gps) and partially supported by the grant CICYT/TAP98-0345 from the Spanish government.

REFERENCES

[1] J.C. Beck, *A Schema for Constraint Relaxation with Instantiations for Partial Constraint Satisfaction and Schedule Optimization*, Master's Thesis, Technical Report EIT-94-3, University of Toronto, (1994).

[2] J.C. Beck, A.J. Davenport, E.M. Sitarski and M.S. Fox, *Beyond Contention: Extending Texture-Based Scheduling Heuristics*, In National Conference On Artificial Intelligence (AAAI-97), (1997).

[3] R. Dechter, I. Meiri and J. Pearl, 'Temporal constraint networks', *Artificial Intelligence*, **49**, 61-95, (1991).

[4] M.S. Fox and N. Sadeh, *Why is Scheduling difficult? A CSP Perspective*, In 9th European Conference on Artificial Intelligence, (1990).

[5] S. French, *Sequencing and Scheduling: An Introduction to the Mathematics of the Job-Shop*, Wiley, (1982).

[6] V. Kumar, 'Algorithms for Constraint Satisfaction Problems: A Survey' *AI Magazine*, **13(1)**, 32-44, (1992).

[7] T.E. Morton and D.W. Pentico, *Heuristic Scheduling Systems*, Wiley, (1993).

[8] N.M. Sadeh, *Lookahead techniques for micro-opportunistic job-shop scheduling*, Ph.D., CMU-CS-91-102, School of Computer Science, Carnegie Mellon University, Pittsburgh, (1991).

[9] N.M. Sadeh, K. Sycara and Y. Xiong, 'Backtracking techniques for the job-shop scheduling constraint satisfaction problem', *Artificial Intelligence*, **76**, 455-480, (1995).

[10] N.M. Sadeh and M.S. Fox, 'Variable and value ordering heuristics for the job shop scheduling constraint satisfaction problem', *Artificial Intelligence*, **86**, 1-41, (1996).

An accessibility graph learning approach for task planning in large domains

Emmanuel Guéré and Rachid Alami

LAAS-CNRS,
7 avenue du colonel Roche,
31077 Toulouse, France,
{guere, rachid}@laas.fr

Abstract. In the stream of research that aims to speed up practical planners, we propose a new approach to task planning based on Probabilistic Roadmap Methods (PRM). Our contribution is twofold. The first issue concerns an extension of GraphPlan[1] specially designed to deal with "local planning" in large domains. Having a reasonably efficient "local planner", we show how we can build a "global task planner" based on PRM and we discuss its advantages and limitations. The second contribution involves some preliminary results that allow to exploit to domain symmetries and to reduce in drastic manner the size of the "topological" graph. The approach is illustrated by a set of implemented examples that exhibit significant gains.

1 Introduction

Even though task planners have made very substantial progress over the last years, they are still limited in their use. This is the case with large domains where numerous facts and a huge number of possible actions instantiations are not relevant - a posteriori - for solving a given problem.

There are also domains, like in robotics, where the environment has a given topology; learning such a structure will certainly help in building an efficient planner in a given domain. However, the structure of the environment (at least the "useful" one) heavily depends not only on the environment but also on the actions that can be performed. Our aim is to develop a generic planner that will exhibit and learn the "structure" of a given domain. This is the reason why we propose to investigate approaches based on Probabilistic Roadmap (PRM). PRM basically "captures" the space "topology" through random state generation and connectivity tests between states using a local planner. PRM obtains good results in robot path planning because it is relatively easy to test the validity of a randomly generated configuration and because there exist good metrics and numerous very efficient local planners in the configuration space. PRM can even obtain excellent results when careful techniques are devised in order to construct a compact graph and to "direct" the search toward non-explored regions[12].

We propose an extension of these notions to task planning. Our contribution is twofold. The first issue concerns an extension of GraphPlan[1] specially designed to deal with "local planning" in large domains. Having an reasonably efficient "local planner", we show how we can build a first "global task planner" based on PRM and which builds a "topological" graph approximation of the task space.

We also discuss its advantages and limitations. The second contribution involves some preliminary results that allow to exploit domain symmetries and to reduce in a drastic manner the size of the "topological" graph. Both contributions are illustrated through a prototype implementation. The results are very promising.

2 Probabilistic Roadmap Method (PRM) background

2.1 Learning and Using

PRM [9] have been successfully used in path planning. A PRM planner performs in two steps: i) topology learning and ii) using the learned topology to search a solution of a given problem.

PRM builds a graph, $\mathcal{G} = (V, E)$, which "captures" the configuration space topology. The vertices V correspond to randomly generated configurations, and the edges E to the possible connections between vertices. A local planner $\mathcal{L}$ is used to test such a connection. Table 1 shows a basic version of PRM algorithms. The predicate $connect(v, q)$ means that configurations q and v are already connected by the graph. This test allows PRM to avoid cycles; indeed, $\mathcal{G}$ is limited to a tree in order to allow a significantly faster solution search.

To illustrate PRM algorithm, we develop a toy example in figure 1 with $MAX = 5$: PRM chooses randomly the configurations c_1 and c_2. In our example, the local planner $\mathcal{L}$ simply tests the existence of a collision-free straight line between two configurations. $\mathcal{L}$ can not find a path between c_1 and c_2. A new configuration c_3 is randomly generated; PRM creates a connection a between c_1 and c_3 because of $\mathcal{L}(c_1, c_3)$. Adding c_4 creates a new connection b with c_3. Then, c_5 allows to connect c_1 (c) and c_2 (d). When the learning step is stopped, one can use the graph to search for a global solution to a path planning problem. The initial S and goal S' states are first connected to the graph G with the local planner. A search is then performed and obtains a collision-free path $(S \rightarrow c_3 \rightarrow c_1 \rightarrow c_5 \rightarrow c_2 \rightarrow S')$[1].

2.2 A visibility based algorithm

There is clearly a need to limit the size of the graph while maintaining the best possible "coverage". To do so, Move3D[12] proposes a PRM that computes "visibility" roadmaps which consist of two classes of nodes: the *guards* and the *connectors*. When a new valid configuration is randomly found, three cases may arise:

- either it is not visible from any existing guard[2]; it is then added as a new guard to the graph,
- or it is visible by guards belonging to distinct connected components of the current graph; it is then added as a new connector, and the corresponding connected components are merged,
- otherwise, it is visible only by guards belonging to a same connected component; in such case, it is rejected.

[1] We can note that PRM sacrifices optimisation to efficiency. However, once a path is found, various smoothing and optimisation techniques can be used to improve the solution path

[2] A guard [8] corresponds to a node that is able to access all neighbours by $\mathcal{L}$.

$V \leftarrow \phi$; $E \leftarrow \phi$
$Card_E \leftarrow 0$
While $Card_E < MAX$ **do**
  $q \leftarrow Random()$
  If $q \in CS_{free}$ **Then**
    $V \leftarrow V \cup \{q\}$
    $Card_E \leftarrow Card_E + 1$
    $V_c \leftarrow \{v \in V,$ v neighbour of q $\}$
    For each $v \in V_c$(ordered with increasing distance) **do**
      If $\neg connect(v, q) \wedge \mathcal{L}(v, q)$ **Then**
        $E \leftarrow E \cup \{(v, q)\}$
    End For each
End While

Table 1. PRM basic algorithm

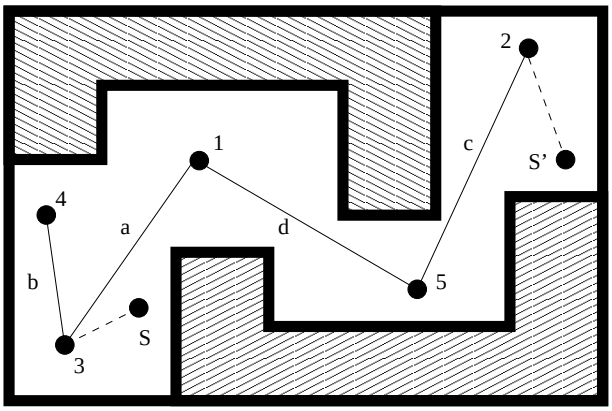


Fig. 1. A PRM sequence example

With such algorithm, c_3 and c_4 of figure 1 would not have been added because of their visibility from c_1.

Following the PRM framework, there is clearly a need for a "very efficient" method for testing states connectivity. There is no need here to have a complete planner. Completeness will be ensured by the global planner (PRM approach). The next section proposes an adaptation of GraphPlan that fulfills such a need.

3 Adapting GraphPlan to local task planning in large domains

In this section, our aim is to develop a "local task planner" to deal with large problem. This local planner will allow to define the "neighbourhood" notion used in PRM. GraphPlan[1] plans with STRIPS operators and uses a constraint propagation method. It performs in two steps: first, it builds a constraint graph expansion; and then it searches for the plan with a constraint resolution extraction. One limitation comes from the number of possible action instantiations and mutex disappearing for large problems[3].

[3] For this section, we assume that the reader is familiar with GraphPlan algorithms.

For instance, RIFO [10] allows IPP to keep only "relevant" facts and to reduce the state size and possible action instantiations. With such state cuts, IPP is able to plan in quite large domains. Our solution is different: we keep the total state description. However in order to deal with the combinatorial explosion due to action instantiation in large domains, we propose to perform a graph expansion based on partial instantiation and we limit the number of levels. This is reasonable since we are interested in developing a fast local planner (in large domains).

To do so we have defined specific tools for forward expansion and backward search in *partial instantiated action* context.

3.1 Partial action instantiation

In STRIPS formalism [4], a state is defined by a set[4] S of positive facts $S = \{f_i\}$. To apply a totally instantiated action B to the state S, its precondition set $P = \{p_i\}$ must be included in S ($P \subseteq S$)[5].

Our goal is to reduce the number of developed actions at each level. We decompose each action B in n partially instantiated actions B_i where $n = Card(P)$.

Definition 1 (Partially instantiated action B_i). *Let $B(\chi)$ be the STRIPS definition of an action B and let χ its arguments (in first order logic) and $P(\chi)$ its preconditions. Let χ_i a partial instantiation of χ such that $p_i(\chi_i)$ is totally instantiated. We call $B(\chi_i)$ (noted B_i) a partial instantiation of B.*

Note that preconditions and effects of B_i are partially instantiated (except for p_i). For instance, table 2 shows a *pick-and-place* action and a partial instantiation by its first precondition.

pick-and-place(x, y, z)	
Precond.:	On(x,y), Clear(x)
	Clear(z)
Eff.:	On(x,z), Clear(y)
	¬ On(x,y), ¬ Clear(z)

pick-and-place($block_1$, $block_2$, z)	
Precond.:	On($block_1$,$block_2$), Clear($block_1$)
	Clear(z)
Eff.:	On($block_1$,z), Clear($block_2$)
	¬On($block_1$,$block_2$), ¬ Clear(z)

Table 2. (a) An example of STRIPS action in block-world domain. (b) A partially instantiated action (by the first precondition On(x,y)).

Definition 2 (B_i Applicability). *B_i is applicable to state S if and only if $p_i(\chi_i) \in S$ and $\forall j \neq i, \exists \chi_j$ (a total instantiation of χ_i) $p_j(\chi_j) \in S$.*

3.2 Mutex propagation

The GraphPlan mutex definition is based on the notion of independence between two actions. Two totally instantiated actions B^1 and B^2 are independent if $B^1 \circ B^2 \Leftrightarrow B^2 \circ B^1$. The independence relation can be defined by Table 3a properties. We can extend this independence relation to partially instantiated actions. B_i^1 and B_ι^2 are independent if $B_i^1 \circ B_\iota^2 \Leftrightarrow B_\iota^2 \circ B_i^1$ with relations defined in Table 3b. Note that the independence relation of partially instantiated actions is weaker than the relation between totally instantiated actions; indeed if $\exists i, \iota$ such that B_i^1 is mutex with B_ι^2 then B^1 is mutex with B^2 but not the converse.

[4] To simplify the notation, we consider states as set of facts instead of conjunctions.

[5] In addition, we have the following properties: i) $D \subseteq P$ and ii) $D \cap A = \phi$ with D (resp. A) the set of facts that become false (resp. true) after applying action B.

$P^1 \subseteq S$	$P^2 \subseteq S$	$\forall j, p_j^1(\chi_j) \in S$	$\forall k, p_k^2(\chi_k) \in S$
$P^1 \cap D^2 = \phi$	$P^2 \cap D^1 = \phi$	$\forall j \exists k, p_j^1(\chi_j) \neq D_k^2(\chi_k)$	$\forall k \exists j, p_k^2(\chi_k) \neq D_j^1(\chi_j)$
$A^1 \cap D^2 = \phi$	$A^2 \cap D^1 = \phi$	$\forall j \exists k, A_j^1(\chi_j) \neq D_k^2(\chi_k)$	$\forall k \exists j, A_k^2(\chi_k) \neq D_j^1(\chi_j)$

Table 3. (a) Independence between B^1 and B^2 in S. (b) Independence between B_i^1 and B_ι^2 (χ_j (resp. χ_k) is a total instantiation of χ_i (resp. χ_ι))

Find_Plan($\mathcal{G}$,$\mathcal{P}$,$\mathcal{E}_{init}$)
  If $\mathcal{G} = \phi$ **Then**
    Return *OK*
  Unstack (X, l_X) from $\mathcal{G}$
  If $l_X = 0$ **Then**
    If $\exists \chi$ an instantiation of X, $\chi \in \mathcal{E}_{init}$ **Then**
      Return Find_Plan($\mathcal{G}$,$\mathcal{P}$,$\mathcal{E}_{init}$)
    Else Return *Fail*
  Else
    Given β the set of partially instantiated action that
    supports X and with $l_\beta \leq l_X$
    Result $\leftarrow$ *Fail*
    While *Result* = *Fail* **do**
      Choose $\{B_i \in \beta\}$such that X is totally instantiated and
      B_i compatible with $B_j, (i \neq j)$ and with $\mathcal{P}$ (non-mutex)
      Add composition of B_i preconditions to $\mathcal{G}$
      Add $\{B_i\}$ to $\mathcal{P}$
      Result $\leftarrow$ Find_Plan($\mathcal{G}$,$\mathcal{P}$,$\mathcal{E}_{init}$)
Return *Result*

Table 4. Plan extraction algorithm for partially instantiated GraphPlan. (X corresponds to a partially or totally instantiated fact, l_X to the level in which we need X and l_β the level of β appearance.)

Nevertheless using partially instantiated actions allows us to generalise the mutex relation between two facts. In GraphPlan, two facts P and Q are mutex if $P = \neg Q$ or if all actions that support P are mutex with all actions that support Q. As we already mentioned, some effects of a partially instantiated action are partially instantiated, and so we can create mutex between two "classes" of facts.

For instance, consider the *pick-and-place* action and a *no-op* action on fact $Clear(block_3)$. The *pick-and-place* action instantiated by the third precondition ($Clear(block_3)$) deletes $Clear(block_3)$ and adds $On(x, block_3)$, whereas *no-op* maintains $Clear(block_3)$. These two actions are mutex, and so we can conclude that $\forall x \in \{block_1, \ldots, block_m\}$, $On(x, block_3)$ is mutex with $Clear(block_3)$.

3.3 Solution extraction

Our planner uses the partial action instanciation to expand the mutex graph starting from the initial state. As in GraphPlan, we try to extract a solution as soon as we reach a level that includes the goal. During backward, the planner must find total instantiations for the selected actions. Such a problem has strong similarities with the extensions of GraphPlan to conformant planning (see CGP[13] and [6]). Indeed, we have adapted the algorithm presented in [6] to deal with partially instantiated actions. Table 4 provides a high level description of the plan extraction procedure.

Block-world domain						
	IPP-v4.0			Our planner (PIGP)		
Problem	Level	CPU Time	Memory	Level	CPU Time	Memory
4 blocks	6	0.1 s	0.4 Mb	6	0.3 s	2.4 Mb
5 blocks	8	0.2 s	0.6 Mb	8	2.4 s	2.5 Mb
6 blocks	10	0.3 s	0.8 Mb	10	6.5 s	2.7 Mb
10 blocks	3	5.8 s	5.7 Mb	3	0.2 s	2.9 Mb
20 blocks	3	269.3 s	95 Mb	3	2.9 s	5.1 Mb
30 blocks	-	>1000 s	>200 Mb	3	5.3 s	7.9 Mb
100 blocks	-	-	-	3	37.8 s	15 Mb

Table 5. Results from IPP-v4.0 and our local task planner. The problems (10-100) are defined by: at initial state, all blocks are on table; the goal is to obtain several three block towers.

Results Table 5 compares IPP-v4.0 with our GraphPlan adaptation on block-world domain. We note that IPP is significantly faster than our algorithm on small domains. On the other hand our planner can elaborate plans in larger domains when the number of levels necessary to reach the goal is small. Indeed, while the partial instantiation reduces drastically the combinatorial explosion of the graph expansion phase, it is expensive for plan extraction.

This is acceptable in our case since we are interested in developing a fast local planner in large domains.

4 Task planning with PRM

In this section, we describe our adaptation of PRM to task planning. It makes use of the local planner defined in the previous section to compute a "topological graph" of the task space. This is done by randomly generating states and trying to connect them to the graph using the local planner.

The process is stopped when we consider that we have a sufficient coverage of the task space. The result is a domain "skeleton", that will be used as a roadmap.

4.1 Adaptating PRM to task planning

Locality and accessibility Evaluating the distance between two states $d(S, S')$ is NP-Hard. All what we need is an estimation δ of d with $\delta(S, S') \leq d(S, S')$[6].

In our case, to approximate δ, we use the mutex propagation phase of GraphPlan. Indeed the number of developed levels to possibly obtain a state S' from S represents the minimal number of independent action set, and so the minimal number of action.

Definition 3 (Accessibility). *State S' is accessible from S (noted $\mathcal{A}(S', S)$), if and only if there exists an action sequence Γ such that $S' = \Gamma(S)$. The direct accessibility corresponds to the existence of a local plan $\mathcal{L}(S', S)$.*

[6] [2] proposes in HSP to evaluate δ with the minimal number of actions to obtain S' from S without *delete-liste*. In [3], he estimates cost from the initial state and uses it to define an heuristic from any state.

We note that the accessibility relation is reflexive (i.e.$\mathcal{A}(S,S)$), transitive (i.e. $\mathcal{A}(S,S'') \wedge \mathcal{A}(S'',S') \rightarrow \mathcal{A}(S,S')$) but not necessarily symetric (i.e. $\mathcal{A}(S,S') \not\equiv \mathcal{A}(S',S)$).

A first algorithm: basic PRM Table 6 describes the incremental construction of the roadmap. The local planner $\mathcal{L}$ is implemented with a partially instantiated GraphPlan (see previous section) and the distance corresponds to the minimal number of graph levels (here we set the neighboorhood to 3 levels).

For instance, given G_1 and G_2 two disjoint components of the graph $\mathcal{G}$ and a state S. If S is accessible from G_1 (i.e. $\exists g \in G_1, \mathcal{A}(S,g)$) and S is not accessible from G_2 (i.e. $\neg\exists g \in G_2, \mathcal{A}(S,g)$) then we assume that state S does not provide any new information about the task space topology[7].

Following [12], we define the notion of *Accuracy*; it corresponds to the current number of uninteresting states (since the last interest state). The number $1 - 1/Accuracy$ corresponds to an estimation of the probabilistic coverage of the task space.

State validity Our algorithm is based on a random generation of states. While it is quite easy to verify the validity of a given configuration in the path planning domain (no overlapping with obstacles), this is not the case, in general, for task planning. For instance, in the blockword domain, we can not authorize $On(block_1, block_2) \wedge On(block_2, block_1)$. Our planner is not able to check state validity. We assume, that we are able to randomly generate all valid states.

First results with basic PRM Figure 2a shows a graph obtained in 7-block-world with a 95% coverage. The program took 727.2 s to build a graph composed of 1867 nodes (3.1 Mb). In this figure, each state is represented by a dot. Two connected dots mean that there exists a local plan between the corresponding states. The position of a given in the diagram depends on the size of highest stack of the state (radius of the circles from 1 to 7) and the number of the first block of the stack (angle of the supporting segment). We note that the figure is strongly symmetric, especially because of the 7 possible first blocks. Indeed, in block-world domain, for a tower of n blocks, there are $n!$ possible configurations.

Figure 2b presents a graph built for the 3-mail problem with a 95% coverage: a robot must move letters from a table to another in a complex environment. In this example, the environment contains 400 cells which are connected with 160000 facts (e.g. $connect(c_{1,1}, c_{1,2})$). Tables are represented by grey cells and walls by black cells. Our algorithm used 18 Mb and took 386.4 s to build a graph composed of 131 nodes. Note that the position of the nodes depends only on robot position and not on the position of the letters (letters can be left on tables or on the robot). This is the reason why there are a number of neighbour nodes on the figure which are not connected.

From these two (non trivial) examples, we can make two observations. First, our algorithm successfully "captured" a topological structure of the task space derived

[7] We assume that S will again be randomly generated, in the future, to test again possible connections between connected components of $\mathcal{G}$. That is the reason why, we can say that the probabilistic completeness of the method is ensured.

$\mathcal{G} \leftarrow \{\phi\}$
$Accuracy \leftarrow 0$
While $Accuracy < MAX$ **do**
  $S \leftarrow RandomValidState()$
  $found_G \leftarrow \phi$; $found_g \leftarrow EmptyState()$
  $found_{nb} \leftarrow 0$
  For each $G \in \mathcal{G}$ (ordered with increasing distance) **do**
    If $\exists g \in G, \mathcal{A}(S, g)$ **Then**
      If $found_{nb} = 0$ **Then**
        $found_G \leftarrow G$; $found_g \leftarrow g$
        $found_{nb} \leftarrow 1$
      Else
        If $found_{nb} = 1$ **Then**
          Connect S to $found_G$ by $found_g$
        $\mathcal{G} \leftarrow \mathcal{G} - G$
        Connect S to G by g
        $found_{nb} \leftarrow found_{nb} + 1$
  End For each
  If $found_{nb} = 0$ **Then**
    $\mathcal{G} \leftarrow \mathcal{G} \cup \{S, \phi\}$
  If $found_{nb} = 1$ **Then**
    $Accuracy \leftarrow Accuracy + 1$
  Else
    $Accuracy \leftarrow 0$
End While

Table 6. Accessibility based algorithm

from accessibility by local plans. Second, in both domains there are symmetries and possible permutations which unusefully increase the graph.

4.2 A PRM that deals with perm utations

Due to all possible permutations between different states, the previous method to build a task topological graph is not able to capture the domain topology with a polynomial number of states (see for instance the symmetry that appears in figure 2a). To solve this problem, we propose an extension that deals explicitly with permutations.

For example, when there is a permutation between two states S_1 ($On(block_1, block_2) \wedge OnTable(block_2)$) and S_2 ($On(block_2, block_1) \wedge OnTable(block_1)$), we will try to learn the task space topology for only one permutation. In this case, the environment "skeleton" is $On(X, Y) \wedge OnTable(Y)$ and there are two possible substitutions $\sigma(S_1, S_2)$ $\{X/block_1; Y/block_2\}$ and $\sigma(S_2, S_1)$ $\{X/block_2; Y/block_1\}$.

We define $\mathcal{A}^+$ the accessibility relation $\mathcal{A}$ augmented by substitution which is transitive: Given S_1, S_2 and S_1' three states such that $\mathcal{A}(S_1, S_2)$, $\mathcal{A}(S_2, S_1)$ and $\sigma(S_1, S_1')$. In that case, there is a plan Γ to connect S_2 to S_1. Given Γ', the plan Γ modified by the substitution $\sigma(S_1, S_1')$, and S_2' the result of the substitution

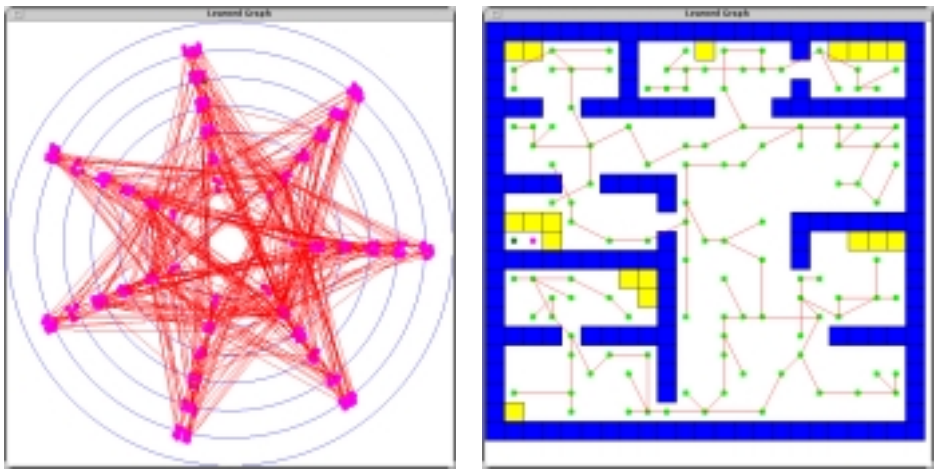


Fig. 2. (a) Learned (95%) graph in 7-block-world domain. (b) Learned (95%) graph in 3-Mail domain.

$\sigma(S_1, S_1')(S_2)$; we have $S_1' = \Gamma'(S_2')$ and we conclude that the accessibility relation $\mathcal{A}(S_1', S_2')$ is valid. So if we have $\mathcal{A}(S_2', S_2)$ then we can deduce $\mathcal{A}(S_1', S_1)$ (*via* the path $S_1 \rightarrow S_2 \rightarrow S_2' \rightarrow S_1'$).

Consider for instance the 3-block-world domain. Given $S_1 = \{\, On(block_1, block_2), On(block_2, block_3), On\_table(block_3) \,\}$, $S_1' = \{\, On(block_3, block_1), On(block_1, block_2), On\_table(block_2) \,\}$ and $S_2 = \{\, On(block_2, block_3), On\_table(block_3), On\_table(block_1) \,\}$. We note that $\neg\mathcal{L}(S_1, S_1')$, $\mathcal{L}(S_1, S_2)$ and $\mathcal{L}(S_2, S_1)$. In addition, we can reach S_2' from S_2 in two steps, so we can conclude that $\mathcal{A}^+(S_1', S_1, \sigma(S_1, S_1'))$[8]

$\mathcal{A}^+(S_1', S_1, \sigma_{1\rightarrow 1'})$
  If $\mathcal{L}(S_1, S_1')$ **Then**
    return *OK*
  For each S_2 such that $\mathcal{L}(S_1, S_2) \wedge \mathcal{L}(S_2, S_1)$ **do**
    $S_2' \leftarrow \sigma_{1\rightarrow 1'}(S_2)$
    If $\mathcal{A}^+(S_2', S_2, \sigma_{1\rightarrow 1'})$ **Then**
      return *OK*
  End For each
  return *Fail*

Table 7. Accessibility based on permutation

This property allows an extension of the basic PRM algorithm that takes into account substitution. The new algorithm is similar to the algorithm presented in table 6 but, instead of using $\mathcal{A}$ to test the accessibility of a new random state S_1, we test if it corresponds to a permutation between two components G_1 and G_2 of the graph $\mathcal{G}$ ($S_2 \in G_2$ such that $\sigma(S_1, S_2)$). If it is the case, we store S_1, S_2 and $\sigma(S_1, S_2)$ and use such permutation to try to connect G_1 and G_2 in the

[8] We note that $\mathcal{A}(S_1', S_1)$ is false because our local planner $\mathcal{L}$ is limited to three steps.

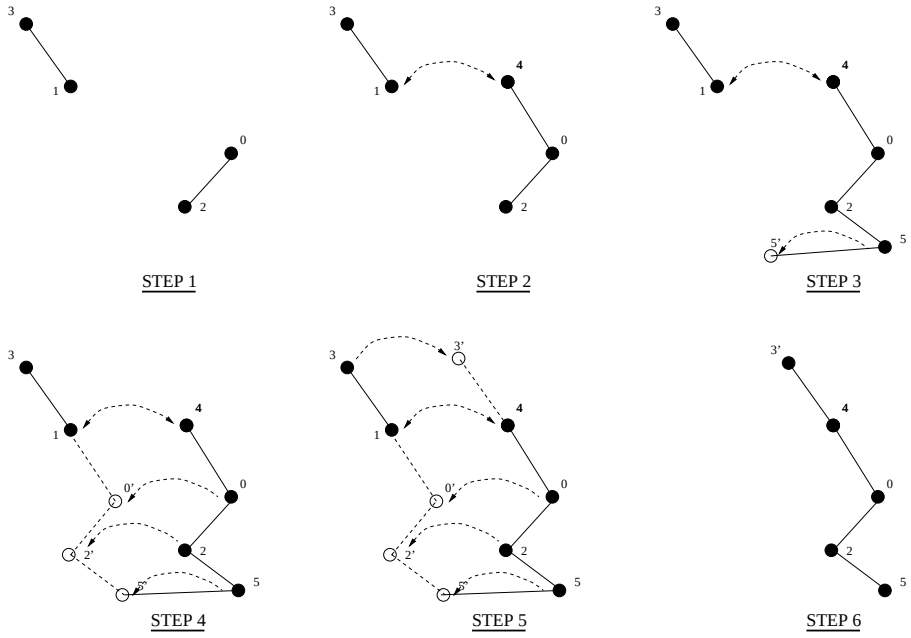


Fig. 3. A PRM sequence exemple

subsquent steps. Besides, if the connection is established, we check the graph in order to eliminate redundancies (see step 6 of figure 3).

The following example illustrates the overall process[9] (see figure 3). At step 1, there are two components in our graph ($S_0 - S_2$ and $S_1 - S_3$). Solid line represent accessibility. At step 2, we randomly generate S_4. S_4 is accessible from S_0 because of $\mathcal{L}$. In addition, we note that there is a substitution $\sigma(S_4, S_1)$. Dashed curve denote a substitution. So the question is: can we connect components $S_4 - S_0 - S_2$ and $S_1 - S_3$? At this step, it is impossible because of: $\neg\mathcal{A}(S_1, S_4)$, $\neg\mathcal{A}(S_3, S_4)$ and not connected at step 1, then we deduce $\neg\mathcal{A}^+(S_1, S_4)$. Now we store $\sigma(S_4, S_1)$ to check if further states can connect the two components via $\mathcal{A}^+$ (this is the case with S_5 and $S_{5'}$). At step 3, we randomly generate S_5. S_5 is accessible from $S_{5'}$ ($S_{5'}$ is created with $\sigma(S_4, S_1)$. In this case (see step 4)), S_4 is accessible from S_1 (indeed we have $S_1 - S_{0'} - S_{2'} - S_{5'} - S_5 - S_2 - S_0 - S_4$). Dashed line means that states are already connected in the other substitution. Now, we can reduce the graph and create only one component without substitution. At step 5, we use $\sigma(S_4, S_1)$ to create $S_{3'}$ from S_3. At step 6, we delete S_1 (resp. S_3) which is now accessible from S_4 (resp. $S_{3'}$) *via* S_5.

Block-world results Figure 4a shows a learned graph for the 7-block-world, with a 99% coverage, domain with permutation method. Our algorithm uses 4.0 Mb during 7.9 s to find 14 nodes Each dot is labelled by a state number described

[9] In order to simplify the explanation we assume that a state accessible from only one component is added to $\mathcal{G}$. For instance in step 1, nodes 2 and 3 are added. We also assume that $\mathcal{A}$ is symmetric.

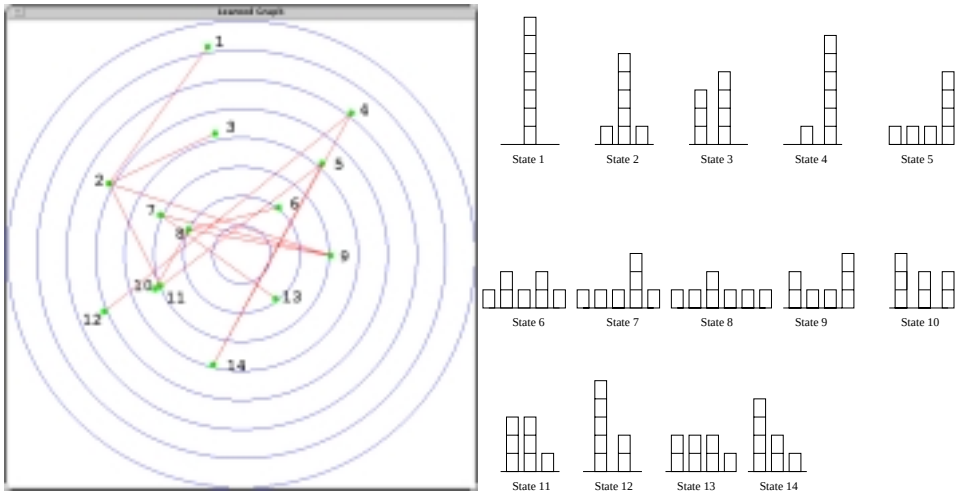


Fig. 4. Learned (99%) graph with substitution method in 7-block-world domain

in figure 4b. We note that for 7 blocks our graph contains only 14 states. These states correspond, in fact, to "classes" of states; indeed because of the permutation reasoning a state of this graph is not really instantiated but represents a whole class of states obtained by substitution.

4.3 Solution search

The solution extraction method is similar to the insertion of one state during the learning phase. Indeed, we must connect the initial state S_{init} to $S_i \in \mathcal{G}$, connect the goal S_{goal} to $S_g \in \mathcal{G}$ and then find a path[10] between S_i and S_g when $\mathcal{A}^+(S_i, S_g)$[11].

Table 8 shows some results on block-world domain (with permutation). Results from 7-block-world express permutation reasoning capabilities: 7.9 s / 14 nodes with 99% *vs.* 727.2 s / 1827 nodes with 95%. In addition, we remark that if we remove the average time spent to connect initial and goal state to the graph from the average time to extract a solution, the spent time to find a path is about 0.1 s.

5 Conclusion and future work

We have proposed an extension of probabilistic Roadmap Methods (PRM) to task planning. Such an extension can not be reasonably attempted without an efficient local planner which may answer quickly to "connections" requests.

This is why we have developed an extension of GraphPlan[1] specially designed to deal with "local planning" in large domains. It is essentially based on the construction of mutex between partially instantiated facts.

[10] We note that the planner is sound. Indeed the solution is extracted from the graph and the connection between the initial and final state; all connections are built by the GraphPlan extension (sound too); so if a path exists, it is consistent.

[11] If we can not connect S_i or S_g we can use a classical planner (e.g. IPP-v4.0).

Problem	Graph learning nb_{node}	CPU Time	Memory	Solution search CPU Time (average)
7 blocks	14	7.9 s	4.0 Mb	0.07 s
10 blocks	31	44 s	4.8 Mb	0.31 s
15 blocks	86	492.5 s	5.6 Mb	1.9 s
20 blocks	253	4186.8 s	6.6 Mb	5.4 s

Table 8. Learn and solution extraction phase on block-world domain

Another key feature is the development of techniques that allow to reduce as much as possible the size of the learned graph without "losing" the probabilistic completeness.

This research is still preliminary. However, the obtained results are very promising. Our future work will concern further investigations on the following aspects: i) improvement of the local planner efficiency (for example, can we introduce some heuristics [3]? ii) improvement of the topological structure identification by adding more general symmetry analysis[5], or extending re-using methods ([11], [7], ...).

References

1. A.L. Blum and M.L. Furst. Fast planning through planning graph analysis. *Artificial Intelligence*, pages 281–300, 1997.
2. B. Bonet and H. Geffner. Hsp: Planning as heuristic search. *http://www.ldc.usb.ve/hector*, 1998.
3. B. Bonet and H. Geffner. Planning as heuristic search: New results. *5th European Conference on Planning (ECP'99)*, 1999.
4. R.E. Fikes and N.L. Nilsson. Strips: A new approach to the application of therom proving to problem solving. *Artificial Intelligence*, 2:189–208, 1971.
5. M. Fox and D. Long. The detection and exploration of symmetry in planning problems. *Proc. 16th Inter. Joint Conf. on Artificial Intelligence (IJCAI'99)*, 1999.
6. E. Guéré and R. Alami. A possibilistic planner that deals with non-determinism and contingency. *Proc. 16th Inter. Joint Conf. on Artificial Intelligence (IJCAI'99)*, 1999.
7. S. Kambhampati and J.A. Hendler. A validation-structure-based theory of plan modification and reuse. *Artificial Intelligence*, 55:193–258, 1992.
8. L. Kavraki, J.-C. Latombe, R. Motwani, and P. Raghavan. Randomized query processing in robot path planning. *Journal of Computer and System Sciences*, 57:50–60, 1998.
9. L. Kavraki, P. Svestka, J.-C. Latombe, and M. Overmars. Probabilistic roadmaps for path planning in high dimensional configuration spaces. *IEEE Transaction on Robotics and Automation*, 12:566–580, 1996.
10. B. Nebel, Y. Dimopoulos, and J. Koehler. Ignoring irrelevant facts and operators in plan generation. In *4th European Conference on Planning (ECP'97)*, 1997.
11. B. Nebel and J. Koehler. Plan reuse versus plan generation: A theorical and empirical analysis. *Artificial Intelligence*, 76:427–454, 1995.
12. T. Simeon, J.P. Laumond, and C. Nissoux. Visibility-based probabilistic roadmaps for motion planning. *(Submitted to Advanced Robotics Journal) A short version appeared in IEEE IROS*, 1999.
13. D. Smith and D. Weld. Conformant graphplan. In *Proc. 15th Nat. Conf. AI.(AAAI'98)*, 1998.

A Heuristic for Domain Independent Planning and its Use in an Enforced Hill-climbing Algorithm

Jörg Hoffmann[1]

Abstract. We present a new heuristic method to evaluate planning states, which is based on solving a relaxation of the planning problem. The solutions to the relaxed problem give a good estimate for the length of a real solution, and they can also be used to guide action selection during planning. Using these informations, we employ a search strategy that combines Hill-climbing with systematic search. The algorithm is complete on what we call *deadlock-free* domains. Though it does not guarantee the solution plans to be optimal, it does find close to optimal plans in most cases. Often, it solves the problems almost without any search at all. In particular, it outperforms all state-of-the-art planners on a large range of domains.

1 INTRODUCTION

The standard approach to obtain a heuristic is to relax the problem $\mathcal{P}$ at hand into some easier problem $\mathcal{P}'$. The optimal solution length to a situation in $\mathcal{P}'$ can then be used as an admissible estimate for the optimal solution length of the same situation in $\mathcal{P}$. An application of this idea to domain independent planning was first used in the HSP system [3]. The planning problem $\mathcal{P}$ is relaxed by simply ignoring the delete lists of all operators. However, computing the optimal solution length for a planning problem without delete lists is still NP-hard, as was first shown by Bylander [4]. Therefore, the HSP heuristic is only a rough estimate of the optimal relaxed solution length. In short, it is obtained by summing up the minimal distances of all atomic goals.

In this paper, we go one step further. We introduce a method that computes *some*, not necessarily optimal, solution to the relaxed problem. These solutions are helpful in two ways:

- their length provides an informative estimate for the difficulty of a situation;
- one can use them as a guidance for action selection.

The solution length estimates are used to control a local search strategy similar to Hill-climbing, which is combined with systematic breadth first search in order to escape local minima or plateaus. The guidance information is employed to cut down the branching factor during systematic search. The method shows good behavior over all domains that are commonly used in the planning community. In particular, we will see that it is complete on the class of problems we call *deadlock-free*. Performing local search, the method can not guarantee its solution plans to be optimal. In spite of this, it finds close to optimal plans in most cases. As a benefit from the severe restriction of its search space, it shows very competitive runtime behavior. For example, *logistics* problems are solved faster than by any other domain independent planning system known to the author at the time of writing.

[1] Institute for Computer Science, Albert Ludwigs University, Georges-Köhler-Allee, Geb. 52, 79110 Freiburg, Germany, email: hoffmann@informatik.uni-freiburg.de

2 BACKGROUND

Throughout the paper, we consider simple STRIPS domains. We briefly review two standard notations. An *action* o has the form

$$o = \langle\ pre(o)\ \Rightarrow\ add(o),\ del(o)\ \rangle$$

where $pre(o)$, $add(o)$ and $del(o)$ are sets of ground facts. Plans P are sequences $P = \langle o_1, \ldots, o_n \rangle$ of actions, i.e., we consider only linear plans.

3 HEURISTIC

In this section, we introduce a method for heuristically evaluating planning states S. Basically, the method consists of two parts.

1. First, the *relaxed fixpoint* is built on S. This is a forward chaining process that determines in how many steps, at best, a fact can be reached from S, and with which actions.
2. Then, a *relaxed solution* is extracted from the fixpoint. This is a sequence of parallel action sets that achieves the goal from S, if their delete effects are ignored.

The first part corresponds directly to the heuristic method that is used in HSP [3]. The second part goes one step further: while in HSP, the heuristic is extracted as a side effect of the fixpoint, we invest some extra effort to find a relaxed plan, and use the plan to determine our heuristic value. The fixpoint process is depicted in Figure 1.

The algorithm can be seen as building a layered graph structure, where fact and action layers are interleaved in an alternating fashion. The process starts with the initial fact layer, which are the facts that are TRUE in S. Then, the first action layer comprises the actions whose preconditions are contained in S. The effects of these actions lead us to the second fact layer, which, in turn, determines the next action layer and so on. The process terminates, and remembers the number max of the last layer, if all goals are reached or if the new fact layer is identical to the last one.

The crucial information that the fixpoint process gives us are the *levels* of all facts and actions. These are defined as the number of the first fact- or action layer they are members of.

$$\text{level}(f) := \begin{cases} min\{i \mid f \in F_i\} & \text{ex. } i : f \in F_i \\ \infty & \text{otherwise} \end{cases}$$

$F_0 := S$
$k := 0$
while $\mathcal{G} \not\subseteq F_k$ **do**
  $O_k := \{o \in \mathcal{O} \mid pre(o) \subseteq F_k\}$
  $F_{k+1} := F_k \cup \bigcup_{o \in O_k} add(o)$
  if $F_{k+1} = F_k$ **then**
    break
  endif
  $k := k + 1$
endwhile
$max := k$

Figure 1. Computing the relaxed fixpoint on a planning state S. $\mathcal{O}$ and $\mathcal{G}$ denote the action set and goal state of the problem at hand, respectively.

$$\text{level}(o) := \begin{cases} min\{i \mid o \in O_i\} & \text{ex. } i : o \in O_i \\ \infty & \text{otherwise} \end{cases}$$

We now show how to extract a relaxed plan from the fixpoint structure. This is done in a backward chaining manner, where we simply use any action with minimal level to make a goal TRUE. The exact algorithm is depicted in Figure 2. Note that we *do not need to search*, we can proceed right away to the initial state and are guaranteed to find a solution.

for $i := 1, \ldots, max$ **do**
  $G_i := \{g \in \mathcal{G} \mid \text{level}(g) = i\}$
endfor
$h := 0$
for $i := max, \ldots, 1$ **do**
  for all $g \in G_i$, g not TRUE at i **do**
    select o with $g \in add(o)$ such that level$(o) = i - 1$
    $h := h + 1$
    for all $f \in pre(o)$, f not TRUE at $i - 1$ **do**
      $G_{\text{level}(f)} := G_{\text{level}(f)} \cup \{f\}$
    endfor
    for all $f \in add(o)$ **do**
      mark f as TRUE at $i - 1$ and i
    endfor
  endfor
endfor

Figure 2. The algorithm that extracts a relaxed solution to a state S after the fixpoint has been built.

Before plan extraction starts, an array of goal sets G_i is initialized by inserting all goals with corresponding level. The mechanism then proceeds down from layer max to layer 1, and selects an action o for each goal g at the current layer i, incrementing the plan length counter h. No actions are selected for goals that are marked TRUE at the time being, as they are already added. The achiever o is required to have level$(o) = i - 1$. This is minimal as the goal g has level i, i.e., the first action that achieved g in the fixpoint came in at level $i - 1$. The preconditions of o are inserted as new goals into their corresponding goal sets. If the current layer is i, then the levels of o's preconditions are at most $i - 1$, so these new goals will be made TRUE later during the process.

3.1 Goal Distance

To obtain the heuristic goal distance value $h(S)$ of a given planning state S, we now simply chain the two algorithms together. First, we perform the fixpoint computation from Figure 1. If the process terminates without reaching the goals, we set $h(S) := \infty$. Otherwise, we extract a relaxed plan, Figure 2, and use the plan length for evaluation, i.e., $h(S) := h$.

The overall structure of the relaxed planning process is quite similar to planning with planning graphs [1]. It amounts to a very special case, as no negative interactions at all occur between facts or actions in the relaxed problem.

3.2 Helpful Actions

We can also use the extracted plan to determine a set of actions that seem to be helpful in reaching the goal. To do this, we turn our look on the actions that are contained in the *first time step* of the relaxed solution, i.e., the actions that are selected at level 0. These are often the actions that are useful in the given situation. Let us see a simple example for that, taken from the *gripper* domain, as it was used in the 1998 AIPS planning systems competition. We do not repeat the exact definition of the domain here, as it is easily understood intuitively. There are two rooms, A and B, and a certain number of balls, which shall be moved from room A to room B. The planner changes rooms via the **move** operator, and controls two grippers which can **pick** or **drop** balls. Each gripper can only hold one ball at a time. We look at a small problem where 2 balls must be moved into room B. A relaxed solution to the initial state that our heuristic might extract is

$<$ { **pick** ball1 A left,
  pick ball2 A left,
  move A B },
 { **drop** ball1 B left,
  drop ball2 B left } $>$

This is a parallel relaxed plan consisting of two time steps. Note that the **move** A B action is selected parallel to the **pick** actions, as the relaxed planner does not notice that it can not **pick** balls in room A anymore once it has moved into room B. In a similar fashion, both balls are picked with the left gripper. Nevertheless, two of the three actions in the first step are helpful in the given situation: both **pick** actions are starting actions of an optimal sequential solution. Thus, one might be tempted to define the set $H(S)$ of helpful actions as only those that are contained in the first time step of the relaxed plan. However, this is too restrictive in some cases. We therefore define our set $H(S)$ as follows.

$$H(S) := \{o \in O_0 \mid add(o) \cap G_1 \neq \emptyset\}$$

After plan extraction, O_0 contains the actions that are applicable in S, and G_1 contains the facts that were goals or subgoals at level 1. Thus, we consider as helpful those actions which add at least one fact that was a (sub)goal at the lowest time step of our relaxed solution.

4 SEARCH

We now introduce a search algorithm that makes effective use of the heuristics we defined in the last section. The key observation that leads us to the method is the following. On some domains, like the *gripper* problems from the 1998 competition and Russel's *tyreworld*, it is sufficient to use our heuristic in a naive *Hill-climbing* strategy. In these problems, one can simply start in the initial state, pick, in each

state, a best valued successor, and ends up with an optimal solution plan. This strategy is very efficient on the problems where it finds plans.

However, the naive method does *not* find plans on most problems. Usually, it runs into an infinite loop. To overcome this problem, one could employ standard Hill-climbing variations, like restarts, limited plateau moves, or a memory for repeated states. We use an *enforced* Hill-climbing method instead, see the definition in Figure 3.

initialize the current plan to the empty plan <>
$S := \mathcal{I}$
obtain $h(S)$ by evaluating S
if $h(S) = \infty$ **then**
  output "No Solution", stop
endif
while $h(S) \neq 0$ **do**
  breadth first search for a state S' with $h(S') < h(S)$
  if no such state can be found **then**
    output "No Solution", stop
  endif
  add the actions on the path to S' at the end of the current plan
  $S := S'$
endwhile

Figure 3. The Enforced Hill-climbing algorithm. $\mathcal{I}$ denotes the initial state of the problem to be solved.

The algorithm combines Hill-climbing with systematic breadth first search. Like standard Hill-climbing, it picks some successor of the current state at each stage of the search. Unlike in standard Hill-Climbing, this successor does not need to be a direct one, and, unlike in standard Hill-Climbing, we do not pick any best valued successor, but *enforce* the successor to be one that is *better* than our current state.

More precisely, at each stage during search a successor state is found by performing breadth first search starting out from the current state S. For each search state S', all successors are generated and evaluated heuristically. Doubly occuring states are pruned from the search by keeping a hashtable of past states in memory, and the search stops as soon as it has found a state S' that has a better heuristic value than S. This way, the Hill-climbing search escapes plateaus and local minima by simply performing exhaustive search for an exit, i.e., a state with strictly better heuristic evaluation.

4.1 Helpful Actions

So far, we have only used the goal distance heuristic. We integrate the helpful actions heuristic into our search algorithm as follows. During breadth first search, we do not generate *all* successors of any search state S' anymore, but consider *only those* that are obtained by applying actions from $H(S')$. This way, the branching factor for the search is cut down. However, the helpful actions heuristic is not completeness-preserving, i.e., considering only the actions in $H(S')$ might make the search miss a goal state. If this happens, i.e., if the search can not reach any new states anymore when restricting the successors to $H(S')$, we simply switch back to complete breadth first search starting out from the current state S and generating *all* successors of search nodes.

5 COMPLETENESS

The Enforced Hill-climbing algorithm is complete on *deadlock-free* planning problems. We define a *deadlock* to be a state S that is reachable from the initial state $\mathcal{I}$, and from which the goal can not be reached anymore. A planning problem is called *deadlock-free*, if it does not contain any deadlock state. We remark that a deadlock-free problem is also solvable, cause otherwise the initial state itself would already be a deadlock.

Theorem 1 *Let $\mathcal{P}$ be a planning problem. If $\mathcal{P}$ is deadlock-free, then the* Enforced Hill-climbing *algorithm, as defined in Figure 3, will find a solution.*

Due to space restrictions, we do not show the (easy) proof of Theorem 1 here and refer the reader to [5]. In short, if the complete breadth first search starting from a state S can not reach a better evaluated state, then, in particular, it can not reach a goal state, which implies that the state S is a deadlock in contradiction to the assumption.

In [5], it is also shown that most of the currently used benchmark domains are in fact *deadlock-free*. Any solvable planning problem that is *invertible* in the sense that one can find, for each action sequence P, an action sequence $\overline{P}$ that undoes P's effects, does not contain deadlocks. One can always go back to the initial state first and execute an arbitrary solution thereafter. Moreover, planning problems that contain an *inverse action* $\overline{o}$ to each action o *are* invertible: simply undo all actions in the sequence P by executing the corresponding inverse actions. Finally, most of the current benchmark domains *do* contain inverse actions. For example in the *blocksworld*, we have **stack** and **unstack**. Similarly in domains that deal with logistics problems, for example *logistics*, *bulldozer*, *gripper* etc., one can often find inverse pairs of actions. If an action is not invertible, its role in the domain is often quite limited. A nice example is the **inflate** operator in the *tyreworld*, which can be used to inflate a spare wheel. Obviously, there is not much point in defining something like a **deflate** operator. More formally speaking, the operator does not destroy a goal or a precondition of any other operator in the domain. In particular, it does not lead into deadlocks.

As one of the anonymous reviewers pointed out to us, deadlock-free domains might be an artificially dominant group because of the simplicity of the current benchmarks. Any domain with consumable resources will contain non-invertible actions. This is certainly true to some extent. We have one theoretical and one practical answer.

- In theory, one can make Enforced Hill-climbing complete on *any* planning problem by simply adding an operator that is applicable in any situation, and reproduces the initial state. That way, search always has the opportunity to go back to the start. In practice, this is not likely to be an effective approach, as it would force complete breadth first search to go all the way down to a state S' with $h(S') < h^{min}$, where h^{min} is the evaluation of the best state seen so far.
- From a more practical point of view, our experience is that Enforced Hill-climbing usually fails quite quickly on the problems which it can not solve. One can then simply switch to a complete heuristic search algorithm, like greedy best-first or weighted A^*.

In the subsequent empirical investigation, we show results for a large collection of benchmark planning problems. All of them but one—a simple *sokoban* instance—are deadlock-free. This is not because we concentrated on solving problems that are deadlock-free, but because there are very few benchmarks available that are not.

Anyhow, Enforced Hill-climbing finds solutions to *all* of these problems, including the *sokoban* instance containing deadlocks.

6 EMPIRICAL RESULTS

For empirical evaluation, we implemented the Enforced Hill-climbing algorithm, using relaxed plans to evaluate states and to determine helpful actions, in C. We call the resulting planning system FF, which is short for FAST-FORWARD planning system. All running times for FF are measured on a Sparc Ultra 10 running at 350 MHz, with a main memory of 256 M Bytes. Where possible, i.e., for those planners that are publicly available, the running times of other planners were measured on the same machine. We indicate run times taken from the Literature in the text. All planners were run with the default parameters, unless otherwise stated in the text, and all benchmark problems are the standard examples taken from the Literature. Some benchmark problems have been modified in order to show how planners scale to bigger instances. We explain the modifications made, if any, in the text. Dashes indicate that the corresponding planner failed to solve that problem within half an hour.

6.1 The *logistics* Domain

This is a classical domain, involving the transportation of packets via trucks and airplanes. There are two well known test suites. One has been used in the 1998 AIPS planning systems competition, the other one is part of the BLACKBOX distribution. The problems in the competition suite are very hard. In fact, they are so hard that, up to date, no planner has been reported to solve them all. FAST-FORWARD is the first one that does. See Figure 4, showing also the results for GRT [12] and HSP-r [2], which are—as far as the author knows—the two best other domain independent *logistics* planners at the time being.[2]

	HSP-r		GRT		FF	
problem	time	steps	time	steps	time	steps
prob-01	0.36	35	0.28	30	0.06	27
prob-02	3.13	36	1.32	34	0.19	32
prob-03	25.45	64	5.55	60	0.71	54
prob-04	50.13	63	19.28	69	0.98	58
prob-05	0.62	27	0.39	26	0.08	22
prob-06	293.60	83	14.39	80	1.95	73
prob-07	6.20	37	1.76	37	0.38	36
prob-08	-	-	16.37	48	2.04	41
prob-09	371.03	97	50.48	98	2.08	91
prob-10	287.64	121	23.13	117	3.20	103
prob-11	4.58	34	1.54	36	0.21	30
prob-12	-	-	43.06	48	2.01	41
prob-13	-	-	85.58	79	7.73	67
prob-14	-	-	60.20	104	6.97	98
prob-15	19.52	120	67.50	106	1.27	93
prob-16	92.75	69	31.58	62	1.23	55
prob-17	29.35	61	12.19	53	0.63	44
prob-18	-	-	335.05	193	50.76	167
prob-19	-	-	238.98	174	16.26	151
prob-20	-	-	324.12	169	24.40	139
prob-21	-	-	294.23	120	8.93	102
prob-22	-	-	-	-	246.05	282
prob-23	100.67	145	16.86	118	3.84	126
prob-24	-	-	98.54	49	4.17	40
prob-25	-	-	-	-	106.23	181
prob-26	-	-	-	-	71.15	183
prob-27	-	-	-	-	71.26	141
prob-28	-	-	-	-	679.43	265
prob-29	-	-	-	-	589.75	323
prob-30	-	-	-	-	62.4	131

Figure 4. Results of the three domain independent planners best suited for *logistics* problems on the competition suite. Times are in seconds, *steps* counts the number of actions in a sequential plan. For HSP-r, the weighting factor W is set to 5, as was done in the experiments described by Bonet and Geffner in [2].

[2] It is important to distinct the results shown in Figure 4 for HSP-r from those reported by Bonet and Geffner [2]. Those results were taken on the problems from the BLACKBOX distribution, while our results are taken on the competition test suite.

The times for GRT in Figure 4 are from the paper by Refanidis and Vlahavas [12], where they are measured on a Pentium 300 with 64 M Byte main memory. FF outperforms both HSP-r and GRT by an order of magnitude. Also, it finds shorter plans than the other planners.

We also ran FF on the benchmark problems from the BLACKBOX distribution suite, and it solved all of them in less than half a second. Compared to the results shown by Bonet and Geffner [2] for these problems, FF was between 2 and 10 times faster than HSP-r, finding shorter plans in all cases.

6.2 Mixed classical Problems

FAST-FORWARD shows competitive behavior on all commonly used benchmark domains. To exemplify this, we show a table of running times on a variety of different domains in Figure 5, comparing FF against a collection of state-of-the-art planning systems: IPP [8], STAN [9], BLACKBOX [7], and HSP [3].

In Figure 5, the planning problems shown are the following. The *tyreworld* problem was originally formulated by Russell, and asks the planner to replace a flat tire. The problem is modified in a natural way so as to make the planner replace n flat tires. FF is the only planner that is capable of replacing more than three tires, scaling up to much bigger problems.

The *hanoi* problems make the planner solve the well known *Towers of Hanoi* problem, with n discs to be moved. FF also outperforms the other planners on these problems.

The *sokoban* problem encodes a small instance of a well known computer game, where a single stone must be pushed to its goal position. Although the problem contains deadlocks, FF has no difficulties in solving it.

The *manhattan* domain was first introduced by McDermott [10]. In these problems, the planner controls a robot which moves on a $n \times n$ grid world, and has to deal with different kinds of keys and locks. The original problem taken from [10] corresponds to the *mh-11* entry in Tabular 5, where the robot moves on a 11×11 grid. The other entries refer to problems that have been modified to encode 7×7, 15×15 and 19×19 grid worlds, respectively. FF easily handles all of them, finding slightly suboptimal plans.

Finally, the *blocksworld* problems in Figure 5 are benchmark examples taken from [6]. FF outperforms the other planners in terms of running time as well as in terms of solution length.

7 RELATED WORK

The closest relative to the work described in this paper is, quite obviously, the HSP system [3]. In short, HSP does Hill-climbing search, with the heuristic function

$$h(S) := \sum_{g \in \mathcal{G}} weight_S(g)$$

The weight of a fact with respect to a state S is, roughly speaking, the minimum over the sums of the precondition weights of all actions that achieve it. The weights are obtained as a side effect of doing exactly the same fixpoint computation as we do. The main problem in HSP is that the heuristic needs to be recomputed for each single search state, which is very time consuming. Inspired by HSP, a few approaches have been developed that try to cope with this problem, like HSP-r [2] and the GRT-planner [12].

The authors of HSP themselves handle the problem by sticking to their heuristic, but changing the search direction, going backwards

		IPP		STAN		BLACKBOX		HSP		FF	
domain	problem	time	steps	time	steps	time	steps	time	steps	time	steps
tyreworld	fixit-1	0.04	19	0.10	19	0.43	19	0.35	23	0.04	19
tyreworld	fixit-2	11.29	30	1.25	30	114.32	30	-	-	0.09	30
tyreworld	fixit-3	-	-	-	-	933.14	41	-	-	0.20	41
tyreworld	fixit-4	-	-	-	-	-	-	-	-	0.42	52
hanoi	tower-3	0.03	7	0.03	7	0.23	7	0.31	7	0.01	7
hanoi	tower-5	0.11	31	0.27	31	680.6	31	2.04	31	0.09	31
hanoi	tower-7	1.93	127	6.10	127	-	-	23.18	163	0.52	127
hanoi	tower-9	39.31	511	230.20	511	-	-	-	-	6.45	511
sokoban	sokoban-1	1.15	25	1.51	25	1283.29	25	13.87	29	0.22	25
manhattan	mh-7	4.82	35	20.04	35	-	-	1.12	35	0.09	38
manhattan	mh-11	65.12	40	1013.96	40	-	-	13.31	40	0.26	43
manhattan	mh-15	-	-	-	-	-	-	-	-	0.64	59
manhattan	mh-19	-	-	-	-	-	-	-	-	1.53	87
blocksworld	bw-large-a	0.47	10	0.57	10	10.30	10	0.78	11	0.04	7
blocksworld	bw-large-b	2.20	14	4.04	14	160.14	14	1.54	13	0.10	10
blocksworld	bw-large-c	88.17	25	267.08	26	-	-	4.34	20	0.56	16
blocksworld	bw-large-d	362.19	33	-	-	-	-	11.36	27	1.42	20

Figure 5. Running times and quality (in terms of number of actions) of plans for FF and state-of-the-art planners on various classical domains. All planners are run with the default parameters, except HSP, where loop checking needs to be turned on.

from the goal in HSP-r instead of forward from the initial state in HSP. This way, they need to compute a weight value for each fact only once, and simply sum the weights up for a state later during search.

The authors of [12] invert the direction of the HSP *heuristic* instead. While HSP computes distances by going towards the goal, GRT goes from the goal to each fact, and estimates its distance. The function that then extracts, for each state during forward search, the state's heuristic estimate, uses the pre-computed distances as well as some information on which facts will probably be achieved simultaneously.

For the FAST-FORWARD planning system, a somewhat paradoxical extension of HSP has been made. Instead of avoiding the major drawback of the HSP strategy, we even worsen it, at first sight: the heuristic keeps being fully recomputed for each search state, and we even put some extra effort on top of it, by extracting a relaxed solution. However, the overhead for extracting a relaxed solution is marginal, and the relaxed plans can be used to prune unpromising branches from the search tree.

To verify where the enormous run time advantages of FF compared to HSP come from, we ran HSP using Enforced Hill-climbing search with and without helpful actions pruning, as well as FF without helpful actions on the problems from our test suite. Due to space restrictions, we can not show our findings in detail here. It seems that the major steps forward are our variation of Hill-climbing search in contrast to the restart techniques employed in HSP, as well as the helpful actions heuristic, which prunes most of the search space on many problems. Our different heuristic distance estimates seem to result in shorter plans and slightly, about a factor two, better running times, when one compares FF to a version of HSP that uses Enforced Hill-climbing search and helpful actions pruning. We did not yet find the time to do these experiments the other way round, i.e., integrate our heuristic into the HSP search algorithm, as this would involve modifying the original HSP code, which means a lot of implementation work.

There has been at least one more approach in the Literature where goal distances are estimated by ignoring the delete lists of the operators. In [10], Greedy Regression-Match Graphs are introduced. In a nutshell, these estimate the goal distance of a state by backchaining from the goals until facts are reached that are TRUE in the current state, and then counting the estimated minimal number of steps that are needed to achieve the goal state.

To the best of our understanding, the action chains that lead to a state's heuristic estimate in [10] are similar to the relaxed plans that we extract. However, the backchaining process seems to be quite costly. For example, building the Greedy Regression-Match Graph for the initial state of the *manhattan* world 11×11 grid problem is reported to take 25 seconds on a Sparc 2 station. For comparison, we ran FF on a Sparc 4 station. Finding a relaxed plan for the initial state takes less than one hundredth of a second, i.e., the time measured is 0.00 CPU seconds.

The helpful actions heuristic shares some similarities with what is known as relevance from the literature [11]. The main difference is that relevance in the usual sense refers to what is useful for solving the whole problem. Being helpful, on the other hand, refers to something that is useful *in the next step*.

8 CONCLUSION AND OUTLOOK

In this paper, we presented two heuristics for domain independent STRIPS planning, one estimating the distance of a state to the goal, and one collecting a set of promising actions. Both are based on an extension of the heuristic that is used in the HSP system. We showed how these heuristics can be used in a variation of Hill-climbing search, and we have seen that the algorithm is complete on the class of deadlock-free domains. We collected empirical evidence that the resulting planning system is among the fastest planners in existence nowadays, outperforming the other state-of-the-art planners on quite a range of domains, like the *logistics*, *manhattan* and *tyreworld* problems.

To the author, the most exciting question is this: *Why* is the heuristic information obtained in this simple manner so good? It is not really difficult to construct abstract examples where the approach produces arbitrarily bad plans, or uses arbitrarily much time, so why does it almost never go wrong on the benchmark problems? Why is the relaxed solution always so close to a real solution, except for the *Tower of Hanoi* problems? Is it possible to define a notion of "simple" planning domains, where relaxed solutions have desirable properties?

First steps into that direction seem to indicate that, in fact, there might be some underlying theory in that sense. In particular, it can be proven that the Enforced Hill-climbing algorithm finds optimal solutions when the heuristic used is *goal-directed* in the following sense:

$$h(S) < h(S') \;\Rightarrow\; min(S) < min(S')$$

Here, $min(S)$ denotes the length of the shortest possible path from state S to a goal state, i.e., Enforced Hill-climbing is optimal when heuristically better evaluated states are really closer to the goal.

It can also be proven that the length of an *optimal* relaxed solution *is*, in fact, a goal-directed heuristic in the above sense on the prob-

lems from the *gripper* domain that was used in the 1998 planning systems competition. We have not yet, however, been able to identify some general structural property that implies goal-directedness of optimal relaxed solutions.

Apart from these theoretical investigations, we want to extend the algorithms to handle richer planning languages than STRIPS, in particular ADL and resource constrained problems.

ACKNOWLEDGEMENTS

The author thanks Bernhard Nebel for helpful discussions and suggestions on designing the paper. Thanks also go the the referees for their comments which helped improve the paper.

REFERENCES

[1] A. Blum and M. Furst. Fast planning through planning graph analysis. *Artificial Intelligence*, 90(1–2):279–298, 1997.

[2] B. Bonet and H. Geffner. Planning as heuristic search: New results. In *Proceedings of the 5th European Conference on Planning*, pages 359–371, 1999.

[3] B. Bonet, G. Loerincs, and H. Geffner. A robust and fast action selection mechanism for planning. In *Proceedings of the 14th National Conference of the American Association for Artificial Intelligence*, pages 714–719, 1997.

[4] T. Bylander. The computational complexity of propositional STRIPS planning. *Artificial Intelligence*, 69(1–2):165–204, 1994.

[5] J. Hoffmann. A heuristic for domain independent planning and its use in a fast greedy planning algorithm. Technical Report 133, Albert-Ludwigs-University Freiburg, 2000.

[6] H. Kautz and B. Selman. Pushing the envelope: Planning, propositional logic, and stochastic search. In *Proceedings of the 14th National Conference of the American Association for Artificial Intelligence*, pages 1194–1201, 1996.

[7] H. Kautz and B. Selman. Unifying SAT-based and graph-based planning. In *Proceedings of the 16th International Joint Conference on Artificial Intelligence*, pages 318–325, 1999.

[8] J. Koehler, B. Nebel, J. Hoffmann, and Y. Dimopoulos. Extending planning graphs to an ADL subset. In *Proceedings of the 4th European Conference on Planning*, pages 273–285, 1997.

[9] D. Long and M. Fox. Efficient implementation of the plan graph in STAN. *Journal of Artificial Intelligence Research*, 10:87–115, 1999.

[10] D. McDermott. A heuristic estimator for means-ends analysis in planning. In *Proceedings of the 3rd International Conference on Artificial Intelligence Planning Systems*, pages 142–149, 1996.

[11] B. Nebel, Y. Dimopoulos, and J. Koehler. Ignoring irrelevant facts and operators in plan generation. In *Proceedings of the 4th European Conference on Planning*, pages 338–350, 1997.

[12] I. Refanidis and I. Vlahavas. GRT: A domain independent heuristic for strips worlds based on greedy regression tables. In *Proceedings of the 5th European Conference on Planning*, pages 346–358, 1999.

Design and Configuration of Furniture Using Internet-based Virtual Reality Techniques

Bernhard Jung[1] and **Mathias Nousch**[2]

Abstract. BEAVER is a program for a specialized computer aided design (CAD) task: The design of custom-built closets for slanted walls or ceilings. BEAVER is based on standard web technologies only, such as the Virtual Reality Modeling Language (VRML) and Java, which enable BEAVER to be run in a suitable internet browser. A key idea of BEAVER is that the user designs the closet by direct interaction with a visualized 3D model, e.g. when adding shelves or changing the size of the raw construction. In this way, an abstract model of the desired closet can be created and modified very easily and under permanent visual feedback. The knowledge-based system assists the user during this task and ensures that relevant design rules are obeyed. Upon completion of the closet's design, BEAVER automatically configures the abstract model with all fittings required for the assembly of the real closet. Thus, the abstract model is enhanced to a complete specification. This specification is used to generate a shopping list of all needed parts, such as boards and fittings. Also, a customized multimedia assembly manual can be generated on the fly, which guides the user when building the real closet. BEAVER not only demonstrates state-of-the-art capabilities of internet-based virtual reality tools but, also, introduces an easy-to-use, widely available method for customized furniture design: CAD for the rest of us, it's on the web!

Figure 1. Virtual assembly with BEAVER. Shelves can be added to, removed from, or shifted inside the 3D closet model. The program automatically calculates the required size and shape of the shelves, while they are moved to different positions.

1 INTRODUCTION

The immense growth of the World Wide Web (WWW) and the rapid development of related technologies for the presentation of multimedia content enables programmers to create even the most complex applications as integral parts of websites which makes these services reachable for a huge audience of internet users. The most important characteristic features of such applications are their time-, location- and platform-independent availability, (typically) free of charge usage, no cost for production and installation, and little effort for updating the underlying database. These advantages allow service providers to reach customers for whom the use of CD-ROM-based tools is either too large an expenditure or too time-consuming.

BEAVER's specialized application area of customized furniture design was motivated by a cooperation with a German company that produces furniture fittings (connecting fittings, furniture hinges etc.) for the do-it-yourself market. A large amount of their turnover is made in cases where available mass production furniture is suited only insufficiently to the needs of rooms with special characteristics. For example, the conversion of attics with slanted walls into living space calls for individual closets with exactly the roof's angle.

We chose this very specific, yet complex field of furniture design as testbed for the development of a VRML-based virtual construction tool for several reasons: First, the problem area is very narrow and well separable from others. Therefore, a small knowledge base, assisting the user in solving this task could easily be built. And second, the target group of the do-it-yourselfers (DIYer) is highly inexperienced both in designing and assembling of closets as well as in the use of computer aided design tools. Thus, a very intuitive user interface with a short learning phase is required. This makes the application area an excellent test case for verifying the proposed advantages of a web-based virtual design tool as compared to conventional CAD tools.

2 CONVENTIONAL DESIGN AIDS

The correct and complete design of furniture, when tried by DIYers is mostly an exception. The fitting producing company with which we cooperated saw the need, not only to sell the appropriate fittings but also to communicate their proper usage before the sale

[1] Faculty of Technology, AG WBS, University of Bielefeld, PO Box 100131, 33501 Bielefeld, Germany, jung@techfak.uni-bielefeld.de

[2] mathias@nousch.de

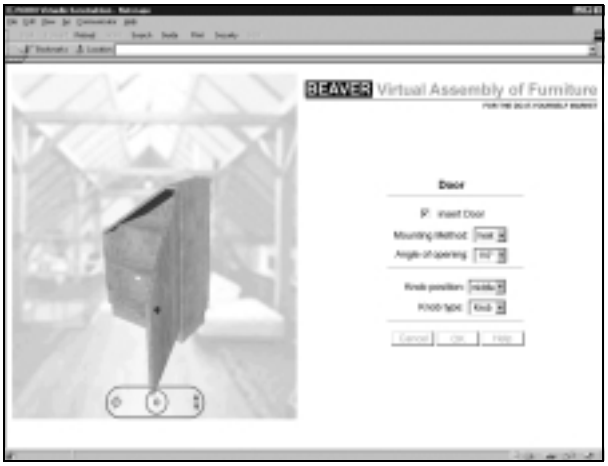


Figure 2. Virtual prototyping allows testing of the model's functions before its real assembly. With BEAVER, for example, the opening angle of the closet can be experienced.

(customers that buy the wrong products are unhappy customers, and potentially no customers at all the next time!).

The most common design aids in this field are printed brochures including a drilling sketch and an assembly instruction for each fitting. The information provided in these brochures may be enough for the assembly of closets from a given furniture construction kit. They are, however, inadequate for the design of completely new closets because they offer no guidance concerning the correct construction of closets as a whole.

The professional furniture manufacturing industry as well as carpenters and interior designers use specialized CAD programs. Such CAD tools are very powerful but also too expensive and too difficult to operate for being used by DIYers.

It was our goal to create a design tool, that would overcome these limitations.

3 CLOSET DESIGN BY VIRTUAL ASSEMBLY

BEAVER pursues the following strategy to solve the given design problem: The user designs an abstract model of the closet through the interactive assembly of virtual components. This is done through the direct manipulation of the visualized 3D model of the closet (see figure 1). The knowledge-based program assists the user during this task by ensuring that all construction rules are met, thus preventing the design of illegal, unbuildable closets. The abstract model is a correct, yet incomplete specification of the desired closet. At the same time, BEAVER translates this abstract specification into a precise one by choosing the appropriate type, number and position of all required fittings and hinges. Once the closet is fully designed, this precise specification is used to generate a shopping list of required parts as well as an individualized assembly manual which can both be used to buy and build the real closet.

BEAVER's multimedia user interface is split into a three-dimensional workroom and a two-dimensional menu. Both, the virtual model inside the three-dimensional workroom and the tabular construction parameters inside the program's menu are visible at any stage of construction. The parallel presentation of the current scene within two different media types helps the inexperienced user in solving emerging ambiguities (cf. [6]).

The multimediality of the display is reflected in the multimodality of interaction: On the one side, the user can *directly* manipulate the virtual model of a component through interaction with its graphical representation. On the other side, the user can manipulate the component *indirectly* by altering the tabular form of its characteristic values. Such alterations are shown immediately inside the 3D-workroom and vice versa. The combination of the two modalities and their dependence upon each other permits a very easy, fast, and communicative introduction to the design process.

In BEAVER, different stages of the closet design process, such as selecting the closet type, setting its size parameters, or adding or modifying shelves, are presented in different frames. This step-by-step approach was chosen because not too much should be expected of an inexperienced user at one time.

One of the most important advantages of virtual prototyping is the possibility to visually experience and verify certain functions of the model before its real counterpart is built. In BEAVER, for example, the hinges are not only static fittings, but consist of two parts, a cross mounting plate and a contained hinge, that can be moved against each other with respect to the hinges' opening angle as shown in figure 2. This enables the user to experience the opening angle of the selected door type and to verify whether this is really the desired angle. In BEAVER, the door can be opened by clicking a button or by dragging the virtual handle of the door.

3.1 Parametric components instead of a static construction kit

The boards used for building a closet are hardly subject to any rules concerning their size or shape. Therefore, in theory, BEAVER has to manage a modular assembly kit of boards that is of infinite size. To prevent this, only specific, but parametric kinds of boards were modeled: Boards are six-sided blocks that can

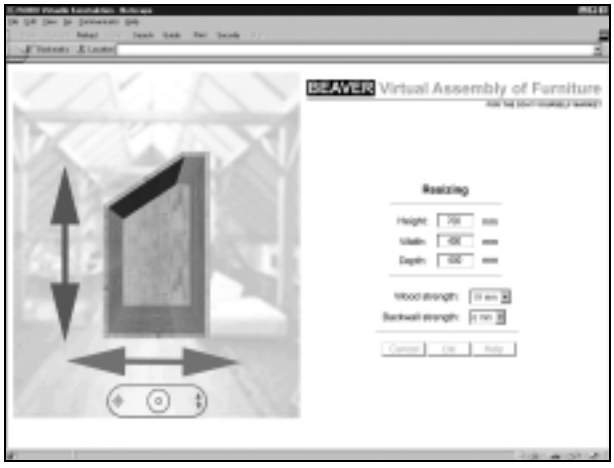


Figure 3. The closet's size can be altered either by typing the absolute values, or by directly dragging the closet's walls. The visual and tabular presentations are updated simultaneously. Any shelves or doors that have already been added to the closet are reshaped and repositioned to fit the new closet geometry.

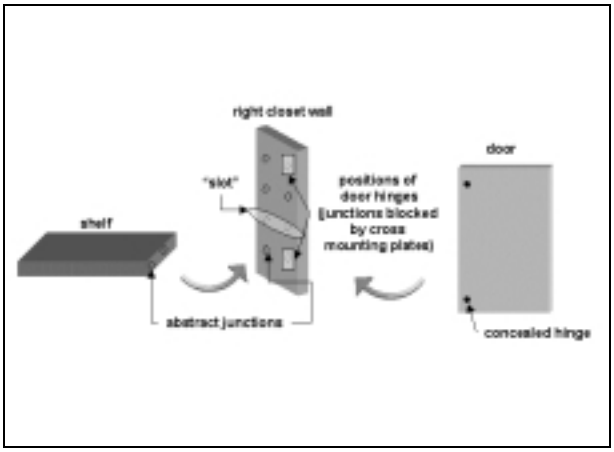


Figure 4. Virtual assembly of closet components is supported by a knowledge-based snapping mechanism. A horizontal pair of junctions forms a *slot* in which a shelf can be placed, if the slot is not already blocked. If the user moves a shelve to a new position, BEAVER automatically snaps it into the nearest free slot.

contain a maximum of two miters at opposing sides. The back or side boards of the closet may be seven-sided, which means they are slanted blocks. Apart from the miters or the slant, every edge is right-angled. The size of the boards however is dynamic and can be altered during the design process. Such alterations are triggered by the user; however, the required calculations are performed by BEAVER.

The user is only allowed to change the parameters of the entire closet, such as height or depth (see figure 3). It is possible to change the size parameters of the whole closet even if other components have already been added to the model, such as shelves or a door. BEAVER then recomputes a new, basic closet model and tries to place all components at their former positions, if possible. The program is also able to resolve conflicts that might occur during this phase (see below, Subsection 3.2).

3.2 Assembling virtual components at abstract junctions

One of the main problems when implementing virtual assembly is an appropriate model of the objects connection possibilities (see [3], [4]). This is somewhat easier if the real counterparts of the virtual components have junctions that can be defined precisely inside the model. Boards, however, have no concrete junctions at all. So we had to assume a fixed number of abstract junctions with a clear location on top of a board's surface. We decided to use the same grid of junctions that is commonly used in the furniture industry to prepare a board with drilling holes for its further use. However, there was still another problem: if certain fittings are attached to these junctions, the shape of the inlaying shelf should, in reality, be modified according to the specific shape of the fitting. Because such a manipulation of the shelf is permanent, no other fitting may be attached to the junction at that location later on. While this is very important for the real assembly task, it would be a great slowing-down, if this behavior were reenacted in our virtual model. At this point, we decided to differentiate between the abstract closet model that the user creates and the detailed closet description that is derived by the program: When the user sizes and positions components, BEAVER only computes Boolean values of the junctions, i.e., whether they are blocked or still free. At this point of the design process, however, BEAVER is not yet concerned with the shelves' detailed shape enabling attachment of fittings.

The placement of shelves in the closet during direct manipulation of the 3D model is supported by a knowledge-based *snapping mechanism* (see [2]). To place the shelf in the closet, the user needs to bring the shelf and the side wall in overlap at their abstract junctions. This overlap does not need to be a complete and precise one, but is supported automatically when the distance between two suitable junctions narrows down below a certain threshold. The snapping mechanism serves as substitute for the lack of precision when positioning the components in the 3D model.

The realization of the snapping mechanism is based on the abstract junctions of the closet's side walls. For the assembly of shelves, two horizontal pairs of junctions at the interior surface of the closet's side walls are needed. We call this a *slot*. A hinge of the closet's door can block one junction of a slot, which prevents the whole slot from being occupied by a shelf, as can be seen in figure 4. During the design of the closet, the user can also shift the shelves in vertical direction. The program keeps track of already used or otherwise blocked slots. When the user drags a shelf with the mouse and drops it, the program interprets this as the command to add this shelf to the nearest empty slot. It then automatically selects the appropriate type and number of fittings for the given shelf-closet combination. If a shelf is dragged to a position where the closet is already slanting, it is obvious, that a normal shelf cannot be placed there. BEAVER recalculates the required shape and size of a shelf, whenever it is repositioned in the closet.

If the size of the closet is modified during the design process, the size and position of all its components must be recalculated. BEAVER uses several heuristics to solve conflicts that can emerge in such situations. For example, if the closet's size is reduced too much, some of its components must possibly be removed. Doors and their hinges are placed with the highest priority. If there is a conflict between a hinge and a shelf, where the latter should be placed in the same slot as the former, the shelf is removed from the model. If there are too many shelves because the closet has been shrunken, only the maximum number of allowed shelves for a certain closet height are retained. If there are no such conflicts, BEAVER tries to position every shelf into the same slot as it was before the change to the closet height was made.

3.3 Configuration, shopping list, and customized assembly instructions

Once the abstract model of the closet consisting of all boards, shelves, and doors has been designed the next step is to configure this abstract model with all fittings and hinges necessary for the assembly of the real closet. As opposed to the interactive design of the abstract closet model, the configuration process is performed automatically by BEAVER. BEAVER's configuration rules ensure that only fittings and hinges suited for this particular piece of furniture are selected. Furthermore – and somewhat unsurprisingly given the commercial interest of our industrial partner – BEAVER only considers products of one manufacturer and, at the request of our industrial partner, products from one product line only which simplifies the configuration process even more.

Figure 5. Fittings and hinges can be examined during the design phase if the model display type is set to transparency. In BEAVER, the user needs not care about which type of fitting he should use or where to place the fittings. BEAVER finds the optimal configuration of fittings for the specific closet designed.

BEAVER's configuration process not only calculates the appropriate kinds and number of fittings and hinges but also their spatial positioning within the closet's 3D model. Algorithmically, the configuration process is very simple. It exploits the fact that - after decade long efforts by the furniture industry of simplification and standardization - there are well-defined rules for equipping furniture with fittings and hinges (these rules are however not usually known by the end user and that's one of the points of a tool like BEAVER): Basically, the type and number of fittings and hinges is very similar for all closets. Closets with doors will need hinges for their attachment, where the number of hinges (two or three) for each door depends on its size. Similarly, closets with a base will need extra fittings and each shelf is attached with four fittings. Variations in the selection of fittings and hinges stem from differences in thickness of the boards and visibility considerations.

Configuration of the closet with fittings and hinges occurs in two phases. The first phase computes number, spatial layout, and product family of the connecting elements but not their concrete model numbers. This first phase is actually interleaved with the design process as e.g. the hinges necessary for the attachment of the doors may conflict with the positioning of the shelves. The positions of fittings and hinges within the closet can be inspected at any stage of the design in a special rendering mode where boards are visualized transparently (see figure 5).

Once the design task is complete, a shopping list of required parts for the real closet is generated. To achieve this, the configuration process enters its second phase. Given the number and product families of connecting elements computed before, exact model types are proposed that e.g. also account for the thickness of the boards used in the assembly. The shopping list contains the types, serial numbers, and number of packages of the fittings and hinges to be purchased. Each fitting is also described with a photograph to make it easier for the customer to find the specific package in the hardware store. Furthermore, each board is presented with a drawing of its shape - annotated with numeric size information - to remind the customer of the different shapes the board can have and to prevent wrong cuts.

Besides the shopping list, BEAVER also generates assembly instructions customized for the particular piece of furniture. In the generated assembly instructions, the installation process of the fittings is described with local drilling sketches (see figure 6). Their global position, which the user could experience during the design process (see figure 5), is now described in textual form that refers to the size of the abstract drilling grid. Here the generated assembly manual supersedes conventional print-media instructions, as it not only provides information about *how* a fitting is fixed, but also *where* with respect to the complete assembly.

The assembly instructions describe difficult installations of fittings in step-by-step drawings. Also, alternative parameters of fittings, especially of hinges, that can be modified even after their installation can be mentioned. The combination of text and graphics helps the DIYer to check his imaginations against the real assembly task and so to prevent errors.

The shopping list and the assembly instructions are presented as "ordinary" web pages that can be bookmarked and revisited a long time after the closet's design. They can, of course, also be printed out on paper. After all, paper still is the more convenient medium for the DIYer when going to the hardware store or when building the real closet in his attic or workroom.

4 IMPLEMENTATION

We decided to rely on three different programming languages for implementing BEAVER on the World Wide Web.

4.1 Knowledge base and design logic: Java

Java is an excellent solution for the development of complex program structures and attractive graphical user interfaces. Java can be interpreted by any modern web browser and is fast enough for most applications. Further, the acceptance of Java applets is very high among internet users. Thus, we chose Java for controlling the design task and for realizing the two-dimensional user interface. The Java applet's classes are structured in a hierarchical order and mirror our top-down view of the design task:

The *applet(-class)* controls BEAVER's main menu and overall control flow of the program. It also establishes a communication

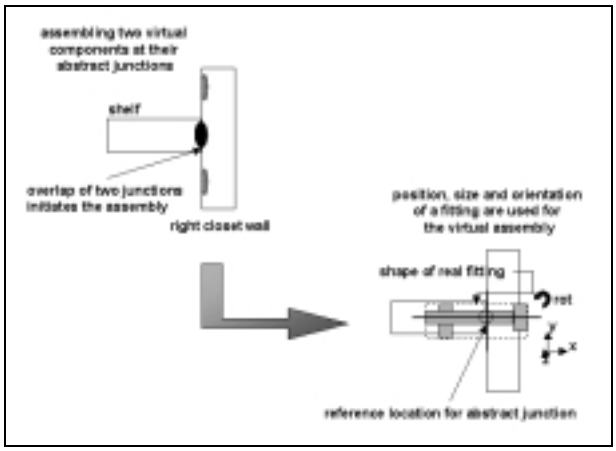


Figure 6. The fittings' detailed shapes are only used in the later stage of generating customized assembly instructions. For the preceding stage of virtual assembly, only basic data of fittings such as position, orientation and size are maintained.

link with the 3D-renderer and provides this link to the other classes.

The class *closet* is the largest class of our applet. It contains the structure of all virtual components, dependencies, and all design and configuration rules modeled in the program. The design and configuration rules are implemented in a straightforward procedural fashion. We favored the procedural approach over a declarative one because the underlying rules for design and configuration are very stable, for efficiency reasons, and to keep the size of the applet (and thus its download time) small. There is only one instantiation of this class, but an enhancement of the applet to allow more than one closet to be built is easily possible. *Closet* is in complete control over the assembly of one closet.

Board exclusively manages the presentation of a closet's components (walls, shelves, door) inside the 3D-workroom. It calculates the numerical shape descriptions of each component to build a virtual three-dimensional object. This class not only controls the appearance of the virtual objects concerning transparency, color, texture, etc. but also handles the user's manipulation of the virtual objects. For example if a component is dragged and released inside the 3D-world this movement is translated into an assembly command and sent to the *closet* class, which in turn decides what further actions to perform.

The class *fitting* is somewhat similar to *board*. It manages the 3D representation of the fitting. This class however does not need any methods for the surveillance of user actions performed on the virtual objects because there are none allowed for fittings. The only dynamic information about fittings maintained by this class are type, function, size, position and orientation. The detailed shape of the physical fitting is also modeled. The detailed shape information serves however only visual purposes and is of no importance for the virtual assembly. As mentioned above, this information is only used in the generation of the assembly instructions.

4.2 Visualization and virtual assembly: VRML

The Virtual Reality Modeling Language (VRML) allows for easy creation of interactive, three-dimensional environments on the WWW. In BEAVER, VRML is used essentially as a 3D renderer with the further possibility of creating „sensible" virtual objects, whose manipulation inside the 3D workroom can be controlled by the applet. The *External Authoring Interface (EAI)* [5] is used to establish the communication link between the VRML browser and the applet that contains the complete design logic.

Our virtual world consists of three different sections: First, there is the main, static VRML file that defines the user's viewpoint and a viewmodel. The viewmodel allows the user to rotate and zoom the view of the designed closet. He cannot navigate freely in the world, because we wanted to prevent inexperienced users from losing sight of the object. The main VRML file also contains the lights and background for the scene.

Second, a navigation console (see figure 7) was developed exclusively for this application. It is realized as an external VRML-File, including a script which generates translation and rotation commands for the main world.

Third, there are the virtual components and fittings of the closet. These are created dynamically by the Java applet via the EAI. For these objects no scripts or routes exist inside the VRML world. All manipulations, even the smallest dragging movements, result from commands that are generated by the applet after it receives input from the VRML sensors. We chose this way of interaction modeling to exclude all possibilities of differences between the internal model of the closet and its external, visual presentation.

4.3 Communication and dynamic page generation: JavaScript

With JavaScript, it is fairly easy to generate dynamic and interactive web pages on the client's side. It is very well suited for the creation of rich multimedia documents. In BEAVER, we use JavaScript to generate the shopping list of required parts and the assembly instructions.

This is done in the following way. The applet breaks down the entire internal representation of the closet into a list of variables and their specific values. Then a new URL on the server is called and the list of parameters is appended to the URL. The HTML file, containing the JavaScript functions reads these parameters and generates the shopping list and assembly instructions. The results are presented as "ordinary" WebPages with text and images, which can be printed or saved to disk. A further advantage of placing the whole parameter list into the URL of the JavaScript page is the following: By just bookmarking this link, the user can save his current design and call the assembly instruction even a long time after the closet was designed. Thus, it is possible to temporarily save the design and continue working on it later.

An important aspect of using JavaScript, as opposed to e.g. signed applets or Perl scripts, is that it avoids the need for writing permission on the client's side or data storage on the server. As future extension, JavaScript'ing may also be used as interface for exporting 3D models of designed closets into CAD programs.

4.4 Notes on development

A fully functional prototype of BEAVER was developed in about

Figure 7. The navigation console is a part of the initial VRML world. In addition to the tabular presentation of the closet's size, a pictograph of a human figure can optionally be displayed in the virtual workroom, lending the user a visual impression of the closet's size.

three months time. This rather small development effort can be attributed to the usage of high-level programming environments such as Java and VRML. The development effort for the prototype also included design reviews by our industrial partners.

Besides providing the mere functionality, a next step was the decision about which products were to be included in BEAVER's database of fittings and hinges. Obviously, this decision was mainly at the side of our industrial partner. Accordingly, the development effort for this second phase was more on the coordination side than on the implementation itself. Interestingly, our industry partner preferred the inclusion of only a rather small number of products into BEAVER's configuration process.

5 CONCLUSIONS

We have described BEAVER, a web-based program for the interactive design of furniture. BEAVER's design methodology draws on multiple knowledge-based technologies to assist the Do-It-Yourselfer in the design process: A snapping mechanism supports the direct manipulation of the visualized 3D model. A configuration process completes the abstract closet model with all required fittings. BEAVER's knowledge of design rules further prevents the construction of unbuildable closets. Besides supporting the mere design of individual closets, BEAVER also generates a shopping list of the required parts and customized multimedia assembly instructions. BEAVER is unique in the combination of these features.

The web-capabilities and free of cost usage of our program were highly welcomed by our industry partner because of BEAVER's potential to enhance the sales of their – but not their competitors - products. Seen from this point, BEAVER is not only an easy-to-use and powerful aid for DIYers for the design and assembly of furniture, but also a somewhat visionary prototype of a new kind of software system for product presentation, promotion, and distribution, possible only on the internet.

BEAVER is specialized on the design of individual closets of non-standard shapes. Its design methodology could also be applied to other design tasks. Considering the interior design industry, other types of furniture could be assembled and brought together in web-based 3D presentation programs such as [1]. BEAVER demonstrates that it is feasible to create specialized CAD tools based on web-based technologies like VRML and Java.

References

[1] J. Dauner, J. Landauer, E. Stimpfig: 3D Product Presentation Online: The Virtual Design Exhibition. *Proceedings of the VRML 98 Third Symposium on the Virtual Reality Modeling Language*, Monterey, CA, ACM press, 1998, p. 57-62

[2] B. Jung, M. Hoffhenke, I. Wachsmuth: Virtual Assembly with Construction Kits. In *Proceedings of the 1998 ASME Design for Engineering* Technical Conferences (DECT-DFM '98).

[3] B. Jung, M. Latoschik, I. Wachsmuth: Knowledge-Based Assembly Simulation for Virtual Prototype Modeling. *IECON'98 – Proceedings of the 24th Annual Conference of the IEEE Industrial Electronics Society*, Vol. 4, IEEE, 1998, 2152-2157.

[4] S. Kopp. *Ein wissensbasierter Ansatz zur Modellierung von Verbindungen zur virtuellen Montage*. Diplomarbeit. Technische Fakultät, Universität Bielefeld, 1998.

[5] C. Marrin and J. Couch. *Specification of the VRML-EAI Interface*, http://www.vrml.org/WorkingGroups/vrml-eai/. The WWW VRML Consortium – Online Resource. 1998.

[6] W. Wahlster. Text and Images. *NSF-CEC Survey on Speech and NL Processing*, 1994

On the Instantiation of ADL Operators Involving Arbitrary First-Order Formulas

Jana Koehler[1] and **Jörg Hoffmann**[2]

Abstract. The generation of the set of all ground actions for a given set of ADL operators, which are allowed to have conditional effects and preconditions that can be represented using arbitrary first-order formulas is a complex process which heavily influences the performance of any planner or pre-planning analysis method.

The paper describes a sophisticated instantiation procedure that determines so-called *inertia* in a given problem representation and uses them to perform simplifications of formulas during the instantiation process. As a result, many inapplicable actions are detected and ruled out from the domain representation yielding a much smaller search space for the planner.

1 Introduction

A planning system that handles a more expressive language than STRIPS requires sophisticated algorithmic solutions to quite a number of problems, which have nothing to do with the actual search process for a plan. One of these problems concerns the computation of the set of *actions* as all ground instances of a given set of operators.

The aim of the instantiation process is to generate all those ground instances of the planning operators that are applicable in some legal world state. This means, that the precondition of the operator should be satisfiable and its effects should be consistent. On one hand, a naive instantiation procedure that simply expands logical quantifiers and enumerates all possible instantiations of operator parameters will quickly render even simple planning problems unsolvable. On the other hand, a rather sophisticated instantiation procedure can rule out many actions, which will never be applicable in any reachable world state or that would—if applied—yield an inconsistent state. It should also return the most simple syntactic representation of preconditions and effects.

Many planning systems do generate the complete set of actions before planning actually starts. They use this set either for the encoding of the domain in other representation formalisms such as SAT [4] or for the derivation of useful information that can help during planning, e.g., distance heuristics [2, 3], symmetries [1], relevant actions [11], and goal orderings [5]. Today, this kind of precomputation appears to be feasible for domains containing up to 100,000 ground actions. But even if only a subset of the available operators is instantiated, e.g., those that are applicable in a given state during a forward search, there is the need for a reasonably well performing instantiation algorithm.

When using the PDDL language [10] to represent ADL operators [12], quite complex descriptions of preconditions and effects are possible:

- arbitrary function-symbol free first-order logic formulas represent preconditions,
- conditional effects have the form *(when antecedent consequent)* where the antecedent can be an arbitrary precondition and the consequent is a conjunction of literals, i.e., an atom either occurs positively or negatively in it. A conditional effect can also be universally quantified.

Given such an operator, the instantiation has to replace all occurring variables, which are either quantified or occur as parameters of the operator, by those type-consistent constants, which have been declared in the planning problem. In order to replace all variables by constants, the instantiation process proceeds in three phases:

1. The expansion of universal and existential quantifiers occurring in the first-order formulas representing preconditions or antecedents of conditional effects eliminates most of the quantified variables,
2. The expansion of universally quantified conditional effects eliminates the remaining quantified variables,
3. the instantiation of operator parameters eliminates the variable parameters.[3]

In each phase, the following *atomic instantiation task* occurs:

> *Given a variable $?x$, a constant c and an atomic formula p, determine the resulting instantiation $p[?x/c]$.*

This is a trivial problem per se. But after having determined $p[?x/c]$ one can sometimes simplify this atomic formula to FALSE or TRUE, which in turn often leads to a further simplification of the operator representation. The paper addresses exactly this problem. We describe what kind of *atomic simplifications* are performed in IPP [7] under which conditions, how this process can be efficiently implemented and how it affects the search space of the planning system. The techniques have successfully been used in the 1998 AIPS planning competition where IPP demonstrated a convincing performance across a variety of STRIPS and ADL domains.

The paper is organized as follows: First, we give an overview of the three phases of the instantiation process. Then we define the notion of *inertia* predicates and describe how the knowledge about in-

[1] Schindler Lifts Ltd., CH-6031 Ebikon, Switzerland, email: jana_koehler@ch.schindler.com

[2] Institute for Computer Science, Albert Ludwigs University, D-79110 Freiburg, Germany, email: hoffmann@informatik.uni-freiburg.de

[3] In IPP the assumption is made that different operator parameters are instantiated with different constants, i.e., the planner never generates actions like *move(a,a)* because we consider this as a bad domain representation that should be revised. In fact, in operators with identical constant parameters, all but one of the constants are superfluous and can be skipped from the representation without loss of information.

ertia is used to perform *atomic simplifications*. We prove their soundness and describe how the underlying tests can be efficiently implemented. In the second part of the paper we define how atomic simplifications can be propagated over the operator description to further simplify the operator representation. We show how unary inertia relations can be encoded as *types* to speed up the instantiation process. Finally, the impact of the instantiation process on the search space of IPP is demonstrated.

2 Overview over the Instantiation Process

After having parsed the domain and problem file into some appropriate data structure, a basic preprocessing step renames all variables in the logical formulas and assigns unique names to them. For example, the formula $\psi(?x) \wedge \forall ?x\, \varphi(?x)$ is equivalently transformed into $\psi(?x_1) \wedge \forall ?x_2\, \varphi(?x_2)$. Then code tables are generated, which map strings to unique numbers, i.e., we obtain one number for each predicate name, variable name, and constant name. Internally, all subsequently described operations work over trees of numbers representing the formulas.

Figure 1 shows the precondition of the **remove** operator from the *assembly* domain [9] with the quantifiers in frames and the underlined *requires* predicate, which will be used throughout this paper to illustrate the instantiation process. This predicate has two arguments, the first one *?whole* being an operator parameter of type *assembly* and the second one *?res* being a universally quantified variable of type *resource*.

```
:action remove
:parameters (?part ?whole - assembly)
:vars (?res - resource)
:precondition
(and ( forall (?res - resource)
        (imply (requires ?whole ?res)
               (committed ?res ?whole)))
     (incorporated ?part ?whole)
     (or (and (transient-part ?part ?whole)
              ( forall (?prev - assembly)
              (imply (remove-order ?prev ?part ?whole)
                     (incorporated ?prev ?whole))))
         (and (part-of ?part ?whole)
              (not ( exists (?prev - assembly)
                   (and (assemble-order ?prev ?part ?whole)
                   (incorporated ?prev ?whole)))))))
```

Figure 1. Precondition of the **remove** operator from the *assembly* domain.

The schematic tree-like representation of this first-order formula is shown in Figure 2. The leaves of the tree contain the atomic formulas. IPP's instantiation process traverses the tree top-down and expands quantifiers one after the other, i.e., it reaches the first quantifier *forall (?res - resource)* and extracts the variable $?res$ together with its type *resource*. From the problem file, IPP knows all constants of this type. These are now used to instantiate $?res$.

The process considers all constants one after the other. For each constant, a copy of the subtree representing the quantified formula is generated. In the leaves of this tree, all occurrences of $?res$ are replaced by the selected constant. As we will see below, this can lead to so-called *atomic simplifications*, which replace an atomic formula by either TRUE or FALSE. In turn, the whole tree can sometimes be simplified to TRUE or FALSE, which yields a dramatic reduction in the size of the formula.

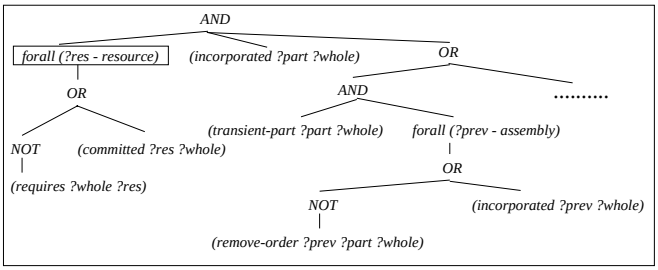


Figure 2. Tree representation of ADL formulas. Note that formulas of the form $\varphi \rightarrow \psi$ have been replaced with the equivalent $\neg\varphi \vee \psi$ already during the parsing process.

In the case of a universal quantifier, the resulting trees are joined by an AND. In the case of an existential quantifier, the trees are joined by an OR. Figure 3 illustrates the result of the process.

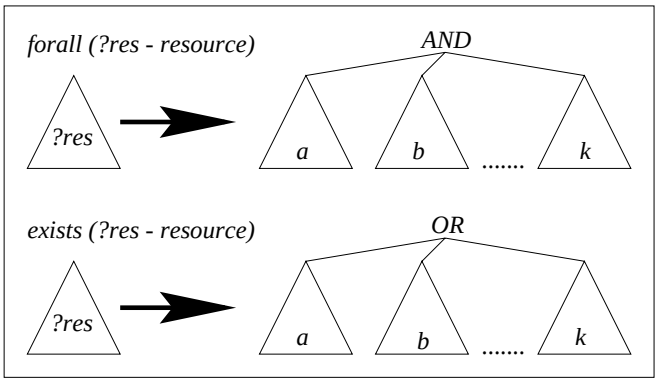


Figure 3. Copies of trees generated during the expansion of quantifiers. Obviously, if one of the subtrees resulting from the expansion of a universal (existential) quantifier can be simplified to FALSE (TRUE), then the whole formula can be simplified to FALSE (TRUE).

The expansion of quantified conditional effects proceeds in a similar way. Figure 4 shows the tree representation of the **move** operator from the *briefcase* domain, whose conditional effect contains the quantifier prefix *forall (?x - object)*. The copied trees will now also contain *when* nodes, i.e., numerous partially instantiated copies of the conditional effect are generated.

The process for instantiating the parameters of an operator fits into the same scheme. In each step, it takes a variable parameter together with the set of type-consistent constants. For each of these constants, a copy of the tree representing the operator is generated, and each occurrence of the parameter in this tree is replaced with the constant. Then, the operator tree is simplified. If, for example, it's precondition simplifies to FALSE, the whole partially instantiated operator can be skipped and removed from the domain. After all parameters have been instantiated, each tree represents a ground instance (i.e., an action) of the operator.

3 Identification of Inertia and their Use during the Instantiation

The tree-copying process takes a variable $?x$ and a constant c as input and traverses the subformula represented in the tree. Whenever it reaches an atomic formula p, it gets replaced with $p[?x/c]$. In many situations, it is worthwhile to invest some more effort at this point

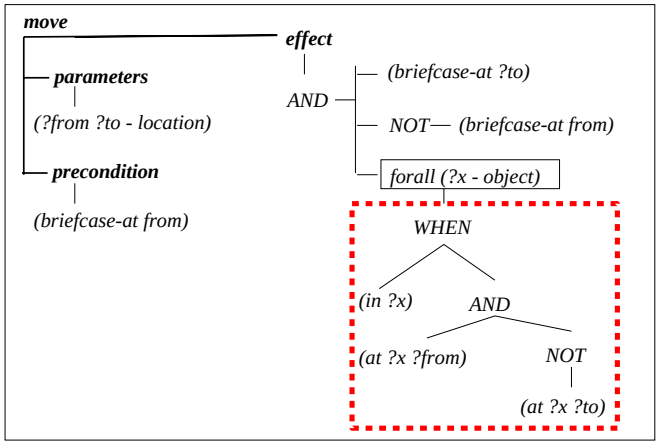


Figure 4. Tree-Representation of an operator with a quantified conditional effect. Expanding the quantifier *forall (?x - object)* results in copies of the tree starting in the *when* node.

and have a closer look at the result of the instantiation. Under certain conditions, namely if p represents an *inertia* relation, one can determine that $p[?x/c]$ must either always be TRUE or FALSE. This can even be the case if $p[?x/c]$ is not yet fully instantiated. Let us consider an example from the *assembly* domain. The object declaration introduces a list of objects followed by their types:

:objects *doodad valve frob sprocket socket plug - assembly*
charger voltmeter battery - resource

The specification of the initial state contains the following instances of the *requires* relation:

:init *(requires frob charger) (requires sprocket charger)*
(requires socket voltmeter) (requires doodad voltmeter)
(requires plug voltmeter)

Given the number of declared constants for the two types, the *requires* relation can be instantiated with $6 \times 3 = 18$ different type-consistent tuples, where 5 of them occur in the initial state.

The expansion of the first universal quantifier that is shown framed in Figure 1 generates three copies of the formula tree, each containing either the partially instantiated atom *(requires ?whole voltmeter)*, *(requires ?whole charger)*, or *(requires ?whole battery)*. Two observations can be made:

- If *(requires ?whole ?res) never occurs as a positive effect* of any operator then the only instances of this predicate, which can hold in any state, are those that are specified in the initial state. This, for example, implies that *(requires ?whole battery)* can never hold and is therefore equivalent to FALSE.
- If *(requires ?whole ?res) never occurs as a negative effect* of any operator then the only instances that can be FALSE in any state are those that are *not* contained in the initial state. Now, if the initial state contained all possible ground instances of, say, *(requires ?whole voltmeter)*, then this partially instantiated predicate could be replaced by TRUE. All of its instances would be initially true and thus persist in all reachable states.

In the following, we will formalize these ideas and give a precise notion of *inertia*.

3.1 Inertia Relations

IPP proceeds over the domain and problem description and collects all used relation names. For each relation it checks if it satisfies one of the following definitions:

Definition 1 *A relation is a* positive inertia *iff it does not occur positively in an unconditional effect or the consequent of a conditional effect of an operator.*

Definition 2 *A relation is a* negative inertia *iff it does not occur negatively in an unconditional effect or the consequent of a conditional effect of an operator.*

Relations, which are positive as well as negative inertia, are simply called *inertia*. Relations, which are neither positive nor negative inertia, are called *fluents*. The detection of inertia and fluents is easy because in ADL, effects are restricted to conjunctions of literals. Furthermore, this information can be obtained with a single pass over the domain description, which takes almost no time at all. In the assembly domain, the status of all relations can be inferred as shown in Figure 5.

predicate name	pos. effect	neg. effect	status
available	yes	yes	fluent
requires	no	no	inertia
part-of	no	no	inertia
transient-part	no	no	inertia
assemble-order	no	no	inertia
remove-order	no	no	inertia
complete	yes	no	neg. inertia
committed	yes	yes	fluent
incorporated	yes	yes	fluent

Figure 5. Inertia Relations in the Assembly Domain.

3.2 Atomic Simplifications

In order to decide if an inertia can be replaced by TRUE or FALSE one needs to determine and count all type-consistent ground instances of an inertia predicate p that match a partially instantiated occurrence of p.

Definition 3 *Let τ be some* type name.

$$dom(\tau) = \{c_1, \ldots, c_m\}$$

denotes the domain *of τ, i.e., the set of constants having type τ.*

In PDDL, each constant is either explicitly declared as being of a particular type or it has the default type *object*. The same applies to all operator parameters or quantified variables. Each predicate must be explicitly declared together with its arguments, for which type names can be given or the default type is assumed.

Definition 4 *Let p be a predicate of arity n. Let $\vec{a} = (a_1, \ldots, a_n)$ be the argument vector of some partially instantiated occurrence of p where each a_i is either a constant or variable. With*

$$V(\vec{a}) = \{i \in \{1, \ldots, n\} \mid a_i \text{ is a variable}\}$$

we denote the positions in $\vec{a}$ that are occupied by variables.

Definition 5 *Let p be a predicate and let $\vec{a}$ be the argument vector of some partially instantiated occurrence of p. Let τ_i be the type name of position i in predicate p. Then*

$$\mathsf{MAX}(p\ \vec{a}) = \prod_{i \in V(\vec{a})} |dom(\tau_i)|$$

denotes the number of all *possible type-consistent ground instances of p that unify with the argument vector $\vec{a}$. In contrast,*

$$\mathsf{N}(p\ \vec{a}) = |\{(p\ \vec{c}) \in \mathcal{I} \mid (p\ \vec{c}) \textit{ unifies with } (p\ \vec{a})\}|$$

denotes the number of unifying ground instances of p that are contained in the initial state $\mathcal{I}$. *Obviously,* $\mathsf{N}(p\ \vec{a}) \leq \mathsf{MAX}(p\ \vec{a})$ *holds.*

It is worthwhile noticing here that IPP will remove all variables or parameters that have an *empty type*. Therefore, we have $|dom(\tau_i)| \neq 0$ for each position i of any partially instantiated occurrence of the predicate p. Thus, for any $(p\ \vec{a})$ we have $\mathsf{MAX}(p\ \vec{a}) \neq 0$. As an example, let us consider $(p\ \vec{a})$ *=(requires ?whole voltmeter)*, for which one obtains

$$\begin{array}{lcl} V(\vec{a}) & = & \{1\} \text{ /* only one variable argument */} \\ \tau_1 & = & \textit{assembly} \\ dom(\textit{assembly}) & = & \{\text{doodad, valve, frob,} \\ & & \text{sprocket, socket, plug}\} \end{array}$$

$\mathsf{MAX}(\textit{requires ?whole voltmeter}) = 6$
/* 6 objects can instantiate *?whole* */
$\mathsf{N}(\textit{requires ?whole voltmeter}) = 3$
/* 3 instances in the initial state contain *voltmeter* */

A partially instantiated atomic formula can be simplified to TRUE or FALSE if it satisfies one of the conditions defined below.

Definition 6 *Let $(p\ \vec{a})$ be some partially instantiated atomic formula constructed during the instantiation process.*

If *p is a positive inertia and* $\mathsf{N}(p\ \vec{a}) = 0$
then *$(p\ \vec{a})$ is simplified to* FALSE.

If *p is a negative inertia and* $\mathsf{N}(p\ \vec{a}) = \mathsf{MAX}(p\ \vec{a})$
then *$(p\ \vec{a})$ is simplified to* TRUE.

In all other cases $(p\ \vec{a})$ cannot (yet) be simplified and remains in the formula tree as it is.

From the treatment of empty types, we know that $\mathsf{MAX}(p\ \vec{a}) \neq 0$ holds for $(p\ \vec{a})$. Therefore, obviously at most one of the above tests can succeed. For example, *(requires ?whole battery)* is a positive inertia. It can be simplified to FALSE because no *requires* instance from the initial state matches the argument vector (*?whole, battery*), i.e., N*(requires ?whole battery)*$= 0$ and the first test succeeds.

That an atomic formula can sometimes be simplified to TRUE is best seen in the case when it is fully instantiated. Take, for example, *(requires plug voltmeter)*. This fact occurs in the initial state, so N*(requires plug voltmeter)*$= 1 \neq 0$ and the first test fails. However, MAX*(requires plug voltmeter)*$= \prod_{i \in \emptyset} |dom(\tau_i)| = 1$ and the second test succeeds. This reflects that *(requires plug voltmeter)* is initially TRUE and will never be made FALSE because *requires* is a negative inertia.

Theorem 1 (Soundness of Simplifications)
Given a planning domain and problem, if $(p\ \vec{a})$ is simplified to
(1) FALSE*, then no state s which is reachable from the initial state satisfies any type-consistent ground instance of $(p\ \vec{a})$.*
(2) TRUE*, then any state s which is reachable from the initial state satisfies all type-consistent ground instances of $(p\ \vec{a})$.*

Proof:
(1) holds because if $\mathsf{N}(p\ \vec{a}) = 0$ then none of the type-consistent ground instances of $(p\ \vec{a})$ are contained the initial state. Since p is a positive inertia, no other instances can be generated by any plan.
(2) holds because if $\mathsf{N}(p\ \vec{a}) = \mathsf{MAX}(p\ \vec{a})$ then all type-consistent ground instances are contained in the initial state and will persist in all reachable states because p is a negative inertia. ■

Atomic simplification requires to determine the number $\mathsf{N}(p\ \vec{a})$ of all those ground tuples in the initial state that unify with a given argument vector of arbitrary length, containing variables or constants at arbitrary positions. Using a naive solution, this means to perform a single pass over the initial state $\mathcal{I}$, testing for each fact if it unifies with $(p\ \vec{a})$. Obviously, the time complexity is $\Theta(|\mathcal{I}| * n_{max})$ where n_{max} denotes the maximum arity of the predicates. The test for atomic simplification has to be done for every leaf of every tree that is ever generated during the instantiation process. The number of these leaves is likely to be enormous, so there is a strong need for a highly efficient method to find $\mathsf{N}(p\ \vec{a})$. In the following, such a method is described, which allows to retrieve the number of matching initial facts in time linear in the *length of* $\vec{a}$, i.e., in the arity of the predicates, $O(n_{max})$.

3.3 Efficient Implementation of Atomic Simplifications

In principle, the idea behind the implementation is as simple as this: Before instantiation starts, perform a single pass over the initial state and create tables in which the occurring tuples are documented. Then later determine the proper table entry for $(p\ \vec{a})$ and look up the correct value of $\mathsf{N}(p\ \vec{a})$. What makes the process complicated is that we have to deal with *partially instantiated* argument vectors $\vec{a}$.

Let us consider the *requires* predicate as an example. For its argument vector of length 2, four cases can occur:

(1) Both arguments are variables and thus $\vec{a} = (?x_1, ?x_2)$. One needs to determine the total number of occurrences of *requires* (with arbitrary arguments) in the initial state.
(2) The first argument is instantiated, but the second argument is a variable and thus $\vec{a} = (c_1, ?x_2)$. We need the number of occurrences of *requires* where the first argument is c_1.
(3) Only the second argument is instantiated and $\vec{a} = (?x_1, c_2)$. We need to count the occurrences with c_2 at the second position.
(4) Both arguments are instantiated and $\vec{a} = (c_1, c_2)$. The question is whether the initial state contains *(requires* c_1 c_2*)*.

For each of these four cases, a separate table is constructed. The table entries are computed from the initial state. The dimension of each table corresponds to the number of instantiated positions of the argument vector. In Case (1), the table is therefore 0-dimensional and simply consists of an integer counting the number of *requires* facts in the initial state. For Cases (2) and (3), a 1-dimensional table is needed, with one entry for each object that is type-consistent with the instantiated argument. For each of these objects, the corresponding entry counts the number of times that *requires* occurred in the initial state instantiated with that object. In Case (4), a 2-dimensional table is constructed. Its entries are indexed by all pairs of type-consistent

objects that can instantiate the *requires* predicate. For each such pair, the entry is set to 1 iff *requires* occurred in the initial state instantiated with that pair. All tables are shown in Figure 6.

$\emptyset$

5

$\{1\}$

doodad	1
valve	0
frob	1
sprocket	1
socket	1
plug	1

$\{2\}$

charge	2
voltmeter	3
battery	0

$\{1,2\}$

	charger	voltmeter	battery
doodad	0	1	0
valve	0	0	0
frob	1	0	0
sprocket	1	0	0
socket	0	1	0
plug	0	1	0

Figure 6. The tables to represent those facts from $\mathcal{I}$ that match a given argument vector for the *requires* predicate. Given the set of instantiated positions as $\emptyset$, $\{1\}$, $\{2\}$ or $\{1,2\}$, the corresponding tables are shown from left to right and down.

Let p be a predicate of arity n. For each subset $C \subseteq \{1,\ldots,n\}$, a table $\mathsf{T}(p\ C)$ has to be constructed. The table is $|C|$-dimensional and lists one entry $\mathsf{T}(p\ C)(\vec{c})$ for each type-consistent tuple $\vec{c}$ of constants that can possibly instantiate p at exactly the positions in C. All entries are initially set to zero.

Note that although the number of tables is exponential in the arity of the predicates, planning domain representations rarely use predicates with more than 3 or 4 arguments. We also argue that it is rather unlikely that significantly more arguments will be required even when more complex domains are modeled. First, the clarity of the representation is affected, which would make it hard for a human user to understand the domain model. Second, it is hard to imagine that an expert in planning domain modeling would set up such a complicated representation. Finally, the few representations of real-world domains, which have been published so far, e.g., [14, 8] show that sources of complexity do not occur necessarily in terms of operator arguments.

Definition 7 *Let $\vec{a} = (a_1,\ldots,a_n)$ be an argument vector of size n, each a_i being either a variable or a constant. Let $C = \{i_1,\ldots,i_k\}$ be a set of possible positions, i.e., $C \subseteq \{1,\ldots,n\}$, where the $i_1,\ldots i_k$ are ordered increasingly. With*

$$\vec{a}|_C := (a_{i_1},\ldots,a_{i_k})$$

we denote the restriction *of $\vec{a}$ to the positions in C.*

Intuitively, the restriction of a vector to some set $C \subseteq \{1,\ldots,n\}$ is obtained by simply skipping all those positions that are not in C, but preserving the order of the arguments.

Now for each ground atom $(p\ \vec{c})$ that occurs in the initial state, the following is done:

for all sets $C \subseteq \{1,\ldots,n\}$ **do**
increment $\mathsf{T}(p\ C)(\vec{c}|_C)$ (1)
endfor

Performing process (1), we count for all instances of p in the initial state how often each combination of constants occurs for arbitrary sets C of positions.

Definition 8 *Let p be a predicate of arity n. Let $\vec{a} = (a_1,\ldots,a_n)$ be the argument vector of some partially instantiated occurrence of p. With*

$$C(\vec{a}) := \{i \in \{1,\ldots,n\} \mid a_i \textit{ is a constant}\}$$

we denote the positions where $\vec{a}$ is instantiated.

During instantiation, given a partially instantiated predicate $(p\ \vec{a})$, we determine the set $C(\vec{a})$ of positions where the argument vector is occupied by constants. The appropriate table $\mathsf{T}(p\ C(\vec{a}))$ is the one corresponding to that set. The entry in this table that we want to access is the one indexed by the constants in the current argument vector $\vec{a}$, i.e., by the restriction $\vec{a}|_{C(\vec{a})}$ of $\vec{a}$ to its constants.

Theorem 2 (Soundness of Tables)
Let the tables T be the result of performing process (1) *for each fact in the initial state. Then we have for each partially instantiated predicate $(p\ \vec{a})$:*

$$\mathsf{N}(p\ \vec{a}) = \mathsf{T}(p\ C(\vec{a}))(\vec{a}|_{C(\vec{a})}).$$

Proof:
Per definition, $\mathsf{N}(p\ \vec{a}) = |\{(p\ \vec{c}) \in \mathcal{I} \mid (p\ \vec{c}) \text{ unifies with } (p\ \vec{a})\}|$. We will show for each fact $(p\ \vec{c}) \in \mathcal{I}$: When process (1) works on $(p\ \vec{c})$,

$\mathsf{T}(p\ C(\vec{a}))(\vec{a}|_{C(\vec{a})})$ gets incremented $\Leftrightarrow$ $(p\ c)$ unifies with $(p\ \vec{a})(*)$

As process (1) is performed for each fact in $\mathcal{I}$, the proposition follows directly from $(*)$, which remains to be shown.

$\Rightarrow$: We prove the contraposition. Let $(p'\ \vec{c}')$ be a fact in $\mathcal{I}$ that does *not* unify with $(p\ \vec{a})$. If $p' \neq p$, process (1) never even considers the table $\mathsf{T}(p\ C(\vec{a}))$. Otherwise, one entry in this table gets incremented when the process reaches $C = C(\vec{a})$. But, as $\vec{c}'$ does not unify with $\vec{a}$, there is at least one constant in $\vec{c}'|_{C(\vec{a})}$ that is different from the corresponding constant in $\vec{a}|_{C(\vec{a})}$. Therefore, the table entry in $\mathsf{T}(p\ C(\vec{a}))$ that gets incremented is different from the one for $\vec{a}|_{C(\vec{a})}$.
$\Leftarrow$: Let $(p\ \vec{c}) \in \mathcal{I}$ be a fact that unifies with $(p\ \vec{a})$. When process (1), working on $(p\ \vec{c})$, reaches $C = C(\vec{a})$, the entry $\mathsf{T}(p\ C(\vec{a}))(\vec{c}|_{C(\vec{a})})$ gets incremented. As $\vec{c}$ is a ground instance that unifies with $\vec{a}$, we have $\vec{c}|_{C(\vec{a})} = \vec{a}|_{C(\vec{a})}$, so this entry is exactly $\mathsf{T}(p\ C(\vec{a}))(\vec{a}|_{C(\vec{a})})$. ■

During the instantiation process it remains to find the corresponding table entry in order to determine the correct value of $\mathsf{N}(p\ \vec{a})$. Since constants are internally kept as *numbers* they can in principle be used as indices into a table. However, to directly index into the tables, one would need to define tables of *arbitrary dimension*. Instead, the implementation uses an implicit representation of the tables. The appropriate address is computed by performing a sweep over the argument vector, which takes time $O(n_{max})$. As arities are usually small, this running time is very close to constant anyway.

3.4 Ground Level Inertia

So far we have only considered the *predicates* which are never made true or false by a planning operator. These were used to constrain the instantiation process. Once the set of all actions has been determined, one can similarly define the *ground facts* that are never made true or false by one of the actions.

Definition 9 *A ground fact is a* positive ground inertia *iff it does not occur positively in an unconditional effect or the consequent of a conditional effect of an action.*

Definition 10 *A ground fact is a* negative ground inertia *iff it does not occur negatively in an unconditional effect or the consequent of a conditional effect of an action.*

An initial fact, which is a negative ground inertia, is never made FALSE and thus always satisfied in all reachable world states. It can be removed from the state description. All its occurrences in the preconditions of actions and in the antecedents of conditional effects can be simplified to TRUE.

A fact, which is a positive ground inertia and *not* contained in the initial state, is never satisfied in any reachable world state. All its occurrences in the preconditions of actions and in the antecedents of conditional effects can be simplified to FALSE.

The remaining facts are fluents that, roughly speaking, can possibly change their truth value during the planning process. They are therefore *relevant* to the representation of the planning problem.

Definition 11 *A ground fact is* relevant *iff*

1. *it is an initial fact and not a negative ground inertia, or if*
2. *it is not an initial fact and not a positive ground inertia.*

Using the table which corresponds to the fully instantiated case of the process described in the previous section, one can find all relevant facts by performing a single sweep over the initial state and the effects of all actions.

The simplified actions and the set of all relevant facts are then used by IPP to generate a bitvector representation for all states and actions, where each relevant fact corresponds to a position in a bitvector.

4 Simplification of Operator Representations

As we have already mentioned in the beginning, the instantiation process creates copies of trees representing formulas and operators. These trees can be simplified if one of their subtrees has been simplified to TRUE or FALSE, which can result from the atomic simplifications performed during the instantiation process. As soon as such an atomic simplification has replaced an atomic formula by TRUE or FALSE, the subsequently described non-atomic simplification operations are performed.[4]

$\neg$TRUE	$\equiv$	FALSE	$\neg$FALSE	$\equiv$	TRUE
TRUE $\wedge\, \varphi$	$\equiv$	φ	$\varphi \wedge \varphi$	$\equiv$	φ
FALSE $\wedge\, \varphi$	$\equiv$	FALSE	$\varphi \vee \varphi$	$\equiv$	φ
TRUE $\vee\, \varphi$	$\equiv$	TRUE	$\varphi \wedge \neg\varphi$	$\equiv$	FALSE
FALSE $\vee\, \varphi$	$\equiv$	φ	$\varphi \vee \neg\varphi$	$\equiv$	TRUE

Figure 7. Implemented Simplifications for First-Order Formulas.

[4] They are also performed once directly after having parsed the domain and problem file.

The first-order formulas which represent the preconditions of operators and the antecedents of conditional effects are simplified based on the well-known tautologies as shown in Figure 7. Besides this, IPP implements the following simplifications:

1. If a quantified variable does not occur in the quantified formula, the quantifier is removed, i.e., $\forall\, ?x\ \varphi(?y)$ is simplified to $\varphi(?y)$.[5]
2. If a quantified variable $?x$ has an *unknown type*, which has not been declared in the :types field of the domain file or if it has an *empty type*, for which no constant has been declared in the problem file, then the quantified formula is replaced by TRUE in the case of a universal quantifier and by FALSE in the case of an existential quantifier.
3. An equality between two identical variable names, $?x = ?x$, is simplified to TRUE. An equality between two identical constants, $c_1 = c_1$, is also simplified to TRUE. If the constants are different, i.e., $c_1 = c_2$, the equality is simplified to FALSE. In a fully instantiated formula, all occurrences of equalities have been replaced by TRUE or FALSE.

The simplification of first-order formulas can reduce a whole precondition, antecedent or consequent to TRUE or FALSE. In this case, the operator description can be simplified:

1. If the antecedent of a conditional effect becomes FALSE, the conditional effect is removed from the operator. In this case, the effect is never applicable because it requires FALSE to hold, i.e., the state must be inconsistent.
2. If the antecedent of a conditional effect becomes TRUE, the conditional effect becomes unconditional.
3. If the consequent of a conditional effect becomes TRUE, the conditional effect is removed because it does not lead to any state transition.
4. If the precondition or the unconditional effect of an operator becomes FALSE, the whole operator is removed from the domain.
5. If an operator has only TRUE as its unconditional effect and no conditional effects, then the whole operator is removed.[6]

In the final set of actions, to which no simplifications can be applied anymore, all unconditional effects are merged into a single conjunction of literals and all conditional effects with identical antecedents are merged into a single conditional effect.
IPP also implements various syntax checks that help to develop proper domain representations:

1. An operator is removed (a warning is issued, but planning continues) if an operator parameter is declared using an unknown or empty type.
2. A parameter is removed from the operator description (a warning is issued, but the operator remains in the set) if it is declared, but nowhere used in the preconditions or effects.[7]

IPP aborts the instantiation process if it encounters one of the following situations:

[5] Unused quantified variables will usually not appear in the initial domain description. They can, however, appear as a result of atomic simplifications.

[6] Removing effects or whole operators can possibly turn fluents into inertia, i.e., one could repeat the whole analysis procedure again. However, such a phenomenon was not observed in any planning domain and therefore it seems not worth to invest the effort into such a fixpoint computation.

[7] Just like unused quantifiers, this can also happen as a result of simplifications.

1. A predicate symbol is overloaded. PDDL requires the declaration of predicates, their arity and the types of their arguments. When parsing the domain and problem files, IPP verifies that all occurrences of a predicate meet the declaration.
2. An equality statement occurs in an unconditional effect or in the consequent of a conditional effect.
3. An equality statement has less or more than two arguments.
4. A variable occurs that is neither declared as a parameter nor bound by a quantifier.
5. A constant occurs that has not been declared in the problem file.

5 Encoding Unary Inertia as Types

Many domains, in particular *all* STRIPS domains used in the 1998 AIPS planning competition contain *unary inertia*. These are predicates of arity one, which satisfy Definitions 1 and 2 and thus do not occur in any of the effects. In other words, the set of constants c that can ever (and will always) satisfy $(p\ c)$ is exactly the set of constants occurring as the arguments of the instances of p in the initial state.

Obviously, this set can be seen as the encoding of *type* information because the single variable argument of p can only be instantiated with one of these constants if we want to obtain a possibly satisfiable atomic formula. As a matter of fact, in the STRIPS domains from the planning competition, all unary inertia where intended to provide implicit type information, as there are no explicit types given in classical STRIPS, see Figure 8 for an example.

One can easily make this implicit type information explicit and remove all unary inertia from the domain description. The previously described instantiation process that identifies and simplifies inertia will also achieve the desired simplification of unary inertia, because they are simply a special case wrt. the length of the argument vector. However, doing it this way, the algorithm repeatedly generates copies of formula trees, only to find out that it can remove them immediately afterwards because they use the "wrong objects" in some unary inertia. For example, when instantiating the set of actions for the problem `strips-log-x-9` from the *logistics* domain used in the competition, 55088 actions are generated for which the instantiation procedure needs 527 seconds.

```
:action load-truck
:parameters (?obj ?truck ?loc)
:precondition (and (obj ?obj) (truck ?truck)
                   (location ?loc) (at ?truck ?loc)
                   (at ?obj ?loc))
:effect (and (not (at ?obj ?loc)) (in ?obj ?truck))
```

Figure 8. The **load-truck** operator from the *logistics* domain. Note the untyped parameters and the underlined unary inertia predicates that implicitly encode the type information.

Consequently, there is the need for a further optimization of the instantiation process, which can be achieved through a separate treatment of unary inertia. The optimization, which is described in detail in this section, encodes all the unary inertia *obj*, *city*, *truck*, *airplane*, *location* and *airport* directly as types, which restrict the instantiation possibilities for the arguments of the operators. Running time for this example decreases down to 63 seconds.[8]

We now give a precise notion of how implicit type information can be made explicit. First, for each unary inertia predicate p the new type symbol τ_p for *the type corresponding to* p is introduced.

Definition 12 *Let p be an inertia predicate of arity 1. The type τ_p* corresponding to *p is defined as the type whose domain comprises all constants c for which $(p\ c)$ holds in the initial state $\mathcal{I}$:*

$$dom(\tau_p) = \{c \mid (p\ c) \in \mathcal{I}\}$$

New types can be constructed from other types by intersecting or subtracting from each other the corresponding sets of constants.

Definition 13 *Let τ_1 and τ_2 be type names. Then $\tau_1 \cap \tau_2$ and $\tau_1 \setminus \tau_2$ are new type names. Their domains are defined as:*

$$dom(\tau_1 \cap \tau_2) = dom(\tau_1) \cap dom(\tau_2)$$
$$dom(\tau_1 \setminus \tau_2) = dom(\tau_1) \setminus dom(\tau_2)$$

After having extracted all types τ_p for unary inertia p from the initial state, the type structure of the domain representation is refined with the types τ_p and types that can be constructed from them.

Definition 14 *Let o be some operator and $?x$ be one of its parameters. Let p be a unary inertia. If $(p\ ?x)$ occurs in the preconditions of o or in the antecedent of one of its conditional effects, o is replaced by two new operators $o1$ and $o2$:*

- *In $o1$, the type τ that has been declared for $?x$ is restricted to $\tau \cap \tau_p$ and all occurrences of $(p\ ?x)$ are replaced with* TRUE.
- *In $o2$, the type τ that has been declared for $?x$ is restricted to $\tau \setminus \tau_p$ and all occurrences of $(p\ ?x)$ are replaced with* FALSE.

Similarly, quantified formulas in preconditions or antecedents of conditional effects are replaced.

Definition 15 *Let $\varphi = \forall\ ?x : \tau\ \psi$ be some universally quantified formula containing a unary inertia p with argument $?x$ of type τ. The formula φ is replaced with φ' defined as*

$$\varphi' = \begin{array}{l} \forall\ ?x : \tau \cap \tau_p\ \ \psi[(p\ ?x)/\mathsf{TRUE}] \wedge \\ \forall\ ?x : \tau \setminus \tau_p\ \ \psi[(p\ ?x)/\mathsf{FALSE}] \end{array}$$

Let $\varphi = \exists\ ?x : \tau\ \psi$ be some existentially quantified formula containing a unary inertia p with argument $?x$. Then φ is replaced with φ'

$$\varphi' = \begin{array}{l} \exists\ ?x : \tau \cap \tau_p\ \ \psi[(p\ ?x)/\mathsf{TRUE}] \vee \\ \exists\ ?x : \tau \setminus \tau_p\ \ \psi[(p\ ?x)/\mathsf{FALSE}] \end{array}$$

In the definition above, $\psi[(p\ ?x)/\mathsf{TRUE}]$ and $\psi[(p\ ?x)/\mathsf{FALSE}]$ denote the formulas, which are obtained from ψ if all occurrences of $(p\ ?x)$ have been replaced with TRUE and FALSE, resp.

The soundness of the replacements follows from the observation that under the restriction $\tau \cap \tau_p$ the atomic formula $(p\ c)$ is always TRUE because only constants c are considered which are also in $dom(\tau_p)$. Under the restriction $\tau \setminus \tau_p$ only constants $c \in dom(\tau)$ are considered that are *not* members of $dom(\tau_p)$ and thus $(p\ c)$ is always FALSE.

We formally state the soundness of the replacements for universally quantified formulas.

[8] The instantiation procedure implemented in IPP 3.3 that has been used in the competition is still a bit faster: It needs only 52 seconds for this example. However, this procedure uses a specialized algorithm which is only capable of handling *conjunctive* preconditions, and it generates a total of 62261 actions because no test for ground inertia is performed.

Theorem 3 (Soundness of Type Encodings)
Let p be a unary inertia predicate. Let $\varphi = \forall\, ?x : \tau\ \psi$ be a formula with $(p\ ?x)$ being a subformula of ψ. Let φ' be the formula φ gets replaced with according to Definition 15. Then, for any state s that is reachable from the initial state holds

$$s \models \varphi \ \Leftrightarrow\ s \models \varphi'$$

Proof:
From the definition of τ_p we know that all constants $c \in \tau_p$ occur as arguments of p in the initial state, i.e., $\mathsf{N}(p\ c) = 1$. For those constants $c \notin \tau_p$, we have $\mathsf{N}(p\ c) = 0$. With Definition 6 and Theorem 1, we get for all states s that are reachable from the initial state:

$$(1)\ \ s \models (p\ c)\ \text{ for }\ c \in \tau_p$$

$$(2)\ \ s \not\models (p\ c)\ \text{ for }\ c \notin \tau_p$$

From this, we can immediately conclude for all states s that are reachable from the initial state:

$$(3)\ \ s \models \psi \Leftrightarrow s \models \psi[(p\ ?x)/\mathsf{TRUE}]\ \text{ for }\ c \in \tau_p$$

$$(4)\ \ s \models \psi \Leftrightarrow s \models \psi[(p\ ?x)/\mathsf{FALSE}]\ \text{ for }\ c \notin \tau_p$$

Thus, for any such state s

$$\begin{aligned} s \models \forall ?x : \tau\psi \Leftrightarrow\ & \text{for all } c \in \tau : s \models \psi[?x/c] \\ \Leftrightarrow\ & \text{for all } c \in \tau \cap \tau_p : s \models \psi[?x/c] \ \text{ and} \\ & \text{for all } c \in \tau \setminus \tau_p : s \models \psi[?x/c] \\ (3)\ \text{and}\ (4)\ \Leftrightarrow\ & \text{for all } c \in \tau \cap \tau_p : s \models \psi[(p\ ?x)/\mathsf{TRUE}] \ \text{ and} \\ & \text{for all } c \in \tau \setminus \tau_p : s \models \psi[(p\ ?x)/\mathsf{FALSE}] \\ \Leftrightarrow\ & s \models \forall\ ?x : \tau \cap \tau_p\ \psi[(p\ ?x)/\mathsf{TRUE}] \wedge \\ & \forall\ ?x : \tau \setminus \tau_p\ \psi[(p\ ?x)/\mathsf{FALSE}] \end{aligned}$$

As the last formula is exactly φ' as defined in Definition 15, the proposition follows. ∎

The soundness of the type encoding follows from the fact that the modified operator set with the newly introduced types has exactly the set of ground instances, which is generated by the instantiation procedure using inertia and performing atomic simplifications that we described in the previous section. The soundness of the replacement of existentially quantified formulas follows with similar arguments as in the universally quantified case.

As an example, let us consider the operator from Figure 8 again. As there is no explicitly defined type for any of the three parameters they are assigned the default type *object*. When examining the first parameter *?obj*, IPP finds that it is used in the unary inertia predicate *obj*. Therefore, it generates two copies of the operator, restricts the parameter types according to Definition 14, and performs the corresponding atomic simplification of the unary inertia. The result is shown in Figure 9.

In the first operator, the atom TRUE can obviously be removed from the conjunction, which leads to a simplified precondition. As *all* constants in the STRIPS *logistics* problems are defined to be of type *object*, the domain $dom(object\ \cap\ \tau_{obj}) = dom(\tau_{obj})$ comprises exactly those constants c for which $(obj\ c)$ is contained in the initial state.

In the second operator, the first atomic precondition has been replaced by FALSE as no constant in $dom(object \setminus \tau_{obj})$ can satisfy $(obj\ c)$. Thus, the whole precondition of this operator simplifies to FALSE and it can be removed from the operator set as it will never be applicable. Note that in the case of arbitrary first-order preconditions one cannot usually expect that operators can be removed immediately just after they have been generated.

:action **load-truck**$^{(1)}$
:parameters *(?obj - object* $\cap\ \tau_{obj}$ *?truck ?loc)*
:precondition *(and (*TRUE *)*
(truck ?truck) (location ?loc)
(at ?truck ?loc) (at ?obj ?loc))

:action **load-truck**$^{(2)}$
:parameters *(?obj - object* $\setminus \tau_{obj}$ *?truck ?loc)*
:precondition *(and (*FALSE *)*
(truck ?truck) (location ?loc)
(at ?truck ?loc) (at ?obj ?loc))

Figure 9. Parameters and preconditions of the two new **load-truck** operators, which result from the encoding of the unary inertia predicate *obj* as a type.

Repeating this process for the other two parameters, always the second copy is removed immediately after it has been generated and thus IPP obtains the final representation of the **load-truck** operator, which is shown in Figure 10.

:action **load-truck**
:parameters *(?obj - object* $\cap\ \tau_{obj}$
?truck - object $\cap\ \tau_{truck}$
?loc - object $\cap\ \tau_{location}$*)*
:precondition *(at ?truck ?loc) (at ?obj ?loc))*
:effect *(and (not (at ?obj ?loc)) (in ?obj ?truck))*

Figure 10. The new operator **load-truck**, which results from the encoding of all unary inertia as types and which replaces the original operator representation. This operator is identical with the one that is used in the typed version of this domain.

The encoding of unary inertia as types is one possibility of how type information can be used to reduce the search space of a planner. TIM [1] implements additional sophisticated type analysis methods, but currently limited to STRIPS. The extension of this work to ADL and its combination with the instantiation method that is described in this paper remains a subject of future work.

6 Empirical Results

Many examples could be presented, which nicely illustrate the benefits of an instantiation procedure that takes inertia into consideration. For example, in the *movie* domain used in the planning competition, 5 operators are declared to get snacks: *get-chips*, *get-dip*, *get-pop*, *get-cheese*, *get-crackers*. Each of them has a similar description, of which we only exemplify the *get-chips* operator:

get-chips
:parameters *(?x - chips)*
:precondition
:effect *(have-chips).*

One observes that the parameter $?x$ is not used anywhere in the operator description. If for example, 9 different constants are declared

for each kind of snack, one obtains 9 ground instances of each operator, which are all identical and spam the search space of the planner. In all *movie* problems, the goals are reachable at time step 1, but a plan can only be extracted at time step 2, i.e., a permutation of all actions at time step 1 is performed by the complete search algorithm. Not very surprisingly, this takes almost 3 s in IPP 3.3 on a Sun Ultra 1/170 because 250973 actions must be tried before a solution is found. In contrast to this, when detecting the unused parameter, only one instance is generated for each operator, which dramatically reduces the search space down to 29 actions and thus a plan is found in only 0.06 s.

In the *assembly* domain, operators can be dramatically simplified because they contain so many inertia. For example, the complex precondition shown in Figure 1 uses 7 different predicates, but 5 of them are inertia. This means that each precondition must simplify to a formula only mentioning the fluents *incorporated* and *committed*. For many actions, the precondition reduces to a single atomic formula using only the *incorporated* predicate. IPP 4.0 is thus able to solve some *assembly* problems, while previous versions failed already during the instantiation, see Figure 11 for selected results.

problem	actions	cpu sec.	search space
assem-x-1	114/760	1742.43	64 673 043
assem-x-2	84/882	18.03	848 829
assem-x-3	190/1248	0.83	108
assem-x-6	118/1800	46342.80	1 283 078 957

Figure 11. Performance of IPP on *assembly* problems on a Sun Ultra 1/170.

Column 2 shows the number of generated actions using the instantiation process with inertia compared to the number of all possible actions using naive enumeration. The search space is measured in the number of actions IPP tries until it finds a plan. The solution plans involve between 31 and 38 actions, but require only between 10 and 18 time steps, i.e., they involve quite some parallelism. Several other problems from this domain can be proven as unsolvable.

The determination of *ground* inertia helps IPP to discover information that it would not be able to find if only inertia *predicates* were analyzed. An interesting example of this behavior occurs in the *tower of Hanoi* domain. Given the operator

move*(?disc,?from,?to: disc)*
:precondition *(and (smaller ?to ?disc)*
(on ?disc ?from) (clear ?disc) (clear ?to))
:effect *(and (clear(?from) (on ?disc ?to)*
(not (on ?disc ?from)) (not (clear ?to))

which describes a legal move of discs, one notices that only a smaller disc can be moved onto a larger disc. IPP discovers that *smaller* is an inertia predicate and only generates the appropriate actions. But the action set also contains moves, which take a disc *from* a smaller disc and put it on a smaller disc. Indeed, the operator description says nothing about the relationship between the disc *?from* and the moving disc *?disc*, i.e., a move that takes a disc from a *smaller* disc and puts it on another smaller disc seems to be a legal action.

When performing the analysis of inertia on the ground level, IPP is able to find out that such moves are impossible. It detects that all instances of *(on ?disc ?from)*, where *?disc* is larger than *?from* are never made true by any action, i.e., they are positive inertia, and they do not hold in the initial state. Thus, these facts are unsatisfiable and all preconditions using them can be simplified to FALSE. Since all actions with FALSE as a precondition are removed from the action set, a further reduction of the size of the planning graph is achieved. For example, in the case of 3 discs, 10 out of 48 actions are eliminated. In the case of 8 discs, 140 out of 468 actions are removed. A search space of only 295.535 actions results and the plan of 255 steps is found in only 16 seconds.

7 Conclusion

The generation of the set of all ground actions for a given set of expressive ADL operators is a complex process which heavily influences the performance of any planner or pre-planning analysis method. The implementation comprises more than 5000 lines of C code and is available from the IPP webpage at http:www.informatik.uni-freiburg.de/~ koehler/ipp.html in the release of IPP 4.0. We hope that the instantiation procedure will become a useful part of reusable code that helps other researcher teams in setting up their own planners more quickly and without dealing with the burden of reimplementing the same preprocessing procedures again and again.

REFERENCES

[1] M. Fox and D. Long, 'The detection and exploitation of symmetry in planning problems', in *Proceedings of the 16th International Joint Conference on Artificial Intelligence*, pp. 956–961. Morgan Kaufmann, San Francisco, CA, (1999).

[2] H. Geffner. HSP: A heuristic search planner. web documentation, 1999.

[3] J. Hoffmann, 'A heuristic for domain independent planning and its use in an enforced hill-climbing algorithm', in *12th International Symposium on Methods for Intelligent Systems*, (2000).

[4] H. Kautz and B. Selman, 'Pushing the envelope: Planning, propositional logic, and stochastic search', in *Proceedings of the 14th National Conference of the American Association for Artificial Intelligence*, eds., D. Weld and B. Clancey, pp. 1194–1201. AAAI Press, (1996).

[5] J. Koehler, 'Solving complex planning tasks through extraction of subproblems', in *Proceedings of the 4th International Conference on Artificial Intelligence Planning Systems*, ed., J. Allen, pp. 62–69. AAAI Press, Menlo Park, (1998).

[6] J. Koehler and J. Hoffmann, 'Handling of inertia in a planning system', Technical Report 122, Albert-Ludwigs-University, (1999). available at http://www.informatik.uni-freiburg.de/~ koehler/ipp.html.

[7] J. Koehler, B. Nebel, J. Hoffmann, and Y. Dimopoulos, 'Extending planning graphs to an ADL subset', In Steel [13], pp. 273–285.

[8] J. Koehler and K. Schuster, 'Elevator control as a planning problem', in *Proceedings of the 5th International Conference on Artificial Intelligence Planning and Scheduling*, eds., S. Chien, S. Kambhampati, and C. Knoblock, pp. 331–338. AAAI Press, Menlo Park, (2000).

[9] D. McDermott. Planning competition benchmark problems. web documentation, http://www.cs.yale.edu/users/mcdermott.hmtl, 1998.

[10] D. McDermott et al., *The PDDL Planning Domain Definition Language*, The AIPS-98 Planning Competition Comitee, 1998.

[11] B. Nebel, Y. Dimopoulos, and J. Koehler, 'Ignoring irrelevant facts and operators in plan generation', In Steel [13], pp. 338–350.

[12] E. Pednault, 'ADL: Exploring the middle ground between STRIPS and the Situation Calculus', in *Proceedings of the 1st International Conference on Principles of Knowledge Representation and Reasoning*, eds., R. Brachman, H.J. Levesque, and R. Reiter, pp. 324–332, Toronto, Canada, (1989). Morgan Kaufmann.

[13] S. Steel, ed. *Proceedings of the 4th European Conference on Planning*, volume 1348 of *LNAI*. Springer, 1997.

[14] B. Williams and R. Nayak, 'A reactive planner for a model-based executive', in *Proceedings of the 15th International Joint Conference on Artificial Intelligence*, pp. 1178–1185. Morgan Kaufmann, San Francisco, CA, (1997).

Considering the Dynamic in Knowledge Based Configuration

Ingo Kreuz[1]

Abstract. Learning from previous solutions could be a key for improving both the solution process and the quality of the solution. Unfortunately knowledge (i.e. components) in technical configuration domains changes quickly and learning brings with it a certain amount of conservatism. For example a component that was learned to be good for a certain task should no longer be chosen, if a newer version appears on the market. On the other hand a certain amount of conservatism is often desired since uncontrolled innovation is as a rule also detrimental, i.e. the newer component mentioned above is not obviously better.

This article presents Relevant Knowledge First (RKF) as a method and heuristic respectively. It tries to find a good compromise between conservatism and innovation based on statistical values and aging of knowledge.

1 INTRODUCTION

Technical markets including computers, multimedia or digital cameras change very quickly. Other technical domains use devices from these fast changing domains, such as cars, trains and airplanes. If one tries to put up a configuration system in these domains he or she will be confronted with the problem of fast changing facts in the knowledge bases. As a rule it is difficult to identify knowledge that is no longer necessary, so it is normally impossible to delete such knowledge. As a consequence the knowledge bases become larger and larger, thus slowing down the configuration processes.

Looking at the field of cognitive psychology we can get an idea of how human experts handle large amounts of quickly changing knowledge: We seem to be able to *concentrate* on actual tasks, we can *learn* from doing something successfully and repeatedly, and above all we are able to *forget* things that are no longer relevant – without deleting them.

In his book [1] Anderson calls the "concentration" the "*activation* of a memory trace". It indicates how accessible information is for a current problem, i.e. how fast and with what probability the information can be accessed. Every time we use a memory trace its accessibility increases a little. If a piece of information is needed frequently, its *action potential* increases what we called "*learn*ing" above. The relationship between the quantity of exercise and the access efficiency (e.g. measured as reaction time) results in a power function which is known in cognitive psychology terms as the *power law of learning*. The learning of information counteracts forgetting: If information is not used over a long period of time, its action potential becomes less. Experiments show that forgetting can also be described as a power function, which amongst other things could be explained by the decaying processes of the neural connections.

The concentration on a given task, the learning and forgetting have lead us to a method for assessing the *relevance* of information: RKF (Relevant Knowledge First) is a heuristic or method for the processing of knowledge bases in the field of configuration. It identifies the *relevance* of information for a given problem thereby simulating some kind of learning

- to speed up and to improve the solution process
- and the solution's quality

and forgetting

- to keep the search space small
- and to give "newer" knowledge the chance to prove its worth, avoiding conservatism.

Note: We do not want to simulate the human model described by Anderson. It simply served as a good starting point for the development of RKF. What we want is a measurement for the relevance of information in knowledge bases for a given task. Though the investigations of cognitive psychology gave us valuable ideas, we will leave this field now and introduce our measurement for relevance.

2 PRINCIPLE OF RKF

We recognized that it is useful to assess knowledge during a knowledge-based search process in order to be able to focus on an actual solution process. With this even large knowledge bases can be scanned efficiently. For this assessment there are two deciding factors which correspond to the antagonism between conservatism and innovation: On one hand the knowledge is very probably useful again if it has already been useful for similar tasks. On the other hand new information should be preferred, in order to obtain innovative solutions and avoid conservatism. For the assessment we use "relevance" which is calculated by age and usefulness of knowledge.

With RKF (Relevant Knowledge First) the search for solutions is supported by relevant knowledge being processed preferentially. When knowledge is selected during a solution process, e.g. in order to bring an object into the solution set or to check its consistency, the relevance for all knowledge *in question* is calculated and *one of*

[1] DaimlerChrysler AG, Research and Technology, HPC T721, D-70546 Stuttgart, Germany, email: ingo.kreuz@daimlerchrysler.com

the most relevant objects is used. For subsequent search processes the relevance is increased for successfully used information.

In order to avoid premature convergence on "bad" results, *the* most relevant knowledge is *not* always used. Instead of this only "one of the most relevant" pieces used. This can be achieved with the help of a random generator, whereby the probability for one choice should be proportional to the relevance of the knowledge.

The relevance of a piece of information i is calculated as functions of its age $a_{i,tc}$ (forget) and its usefulness $u_{i,tc}$ (train) for a given task class tc, whereby usefulness compensates for age. The constant c is used as a domain dependent weighting factor between usefulness and age for the relevance. The second constant m serves to synchronize the measurements used for usefulness and time.

$$relevance(a_{i,tc}, u_{i,tc}) = c \cdot forget(m \cdot a_{i,tc}) + (1\text{-}c) \cdot train(u_{i,tc}) \quad (1)$$

The power functions mentioned in the introduction show desirable characteristics for knowledge bases in technical domains. They were therefore a starting point for the "train" and "forget" functions used in our field, which are introduced in sections 4.1 and 4.2.

For each task class the usefulness of information is separately stored so that the relevance for each task class is calculated independently. This method more or less corresponds to the "activation level of memory traces in certain tasks" functions" or in other words a "concentration on a current task". The independent consideration of different tasks prevents conflicting tasks making training of the knowledge base for good solutions impossible. For finding task classes there are two indicators:

The task classes at first result from the combinations of various global optimization objectives, because conflicting objectives would make a knowledge base training for RKF impossible. Therefore if for example the optimization objectives "price" and "performance" are decisive in one domain, the task class can be automatically generated as a result of the combinations of price and performance together with the objectives "high", "low" and "don't care", e.g. "low price / high performance", "don't care the price / high performance" etc.

As a second indicator task classes can be used, which emerge directly from the respective domain. These are for example target groups of customers for configuration systems. The task classes that has been found automatically can therefore be further refined.

As soon as a solution is found, all information which was useful for finding the solution, i.e. all used information, is upvalued. To do this, the solution is assessed and for all used information in the knowledge base its usefulness is increased in relation to this assessment. In this way, information which has lead to good results becomes useful more quickly. The way in which "usefulness" should precisely be defined, depends on the domain. In section 3, we briefly introduce two of our suggested usefulness measurements which can be applied in technical domains.

Information which was seldom or barely useful, becomes irrelevant bit by bit on the basis of its aging. The measurement employed for age also depends on the domain. Section 3 gives some suggestions for the calculation of age.

In order to find good solutions, RKF is used in an optimization loop: As soon as a solution is found it is assessed, the usefulness of the information concerned is increased and the solution process restarts. The random generator that helps in selecting *one of the most relevant* pieces of information, provides the solutions being different. Due to the repeated increase of usefulness that is based on the assessment the optimization loop converges to a global optimum. The loop can be broken as soon as a "good enough" solution has been found or after a given time limit.

Figure 1 shows the knowledge based search algorithm using RKF:

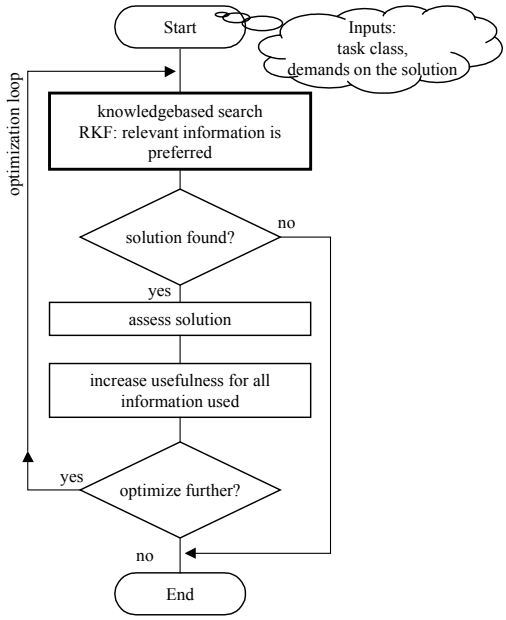


Figure 1. Knowledge based solution process using RKF

3 MEASUREMENTS AND DEFINITIONS FOR THE DETERMINATION OF RELEVANCE

With RKF the relevance of knowledge is calculated as a function of age (time) and usefulness. The measurements for time and usefulness presented in this section serve as a basis for our RKF investigations. In isolated cases they can be further adapted to the area of applicability by which RKF is to be employed

It should be pointed out that relevance of knowledge is only comparable if the same measurements for usefulness and time have been used. Even though several measurements could be applied at the same time. For example a different usefulness measurement can be applied to control knowledge rather than to factual knowledge, in case the distinction between the types of knowledge is made in the knowledge base and the relevance of knowledge of these types is therefore never compared with the other.

Firstly a measurement was sought for the age of information i, which permits comparisons such as "older" and "younger". A relative measurement is therefore sufficient. The age a_i of a piece of information is calculated, as usual, from the difference between the point in time when the information was saved $t_{0,i}$ and the current time t:

$$a_i = t - t_{0,i} \quad (2)$$

Both of the following time measurements have been used in our experiments:

- **RTC** (Real Time Clock): The "real" time of a computer system serves as the current time for the knowledge base.
- **NOR** (Number of Runs): The number of knowledge-based search processes held up to this point in the knowledge base serves as "actual time".

Both measurements have different characteristics. For example using RTC knowledge grows old, even if the knowledge base is not used. If this behavior is not suitable for a domain NOR should be used. An advantage of RTC however is that the semantics of "*outdated knowledge*" is clear i.e. a system's user can, by means of real time, simply decide on the basis of his or her own time feelings.

For the usefulness of information in a knowledge base different definitions can be suitable according to the domain. We have used the following definitions for usefulness measurements:.

- **IPS** (Information was Part of Solution): For each piece of information it is counted how often it was used to help finding solutions. To take the quality of the solutions into consideration the steps are weighted corresponding to the solution's assessment.
- **NAS** (Number of Accesses during Search): This measurement for usefulness also adds up the assessments for the solutions for each piece of information. However in addition to this, the number of times the information in the respective solution process has been accessed is also taken into account.

IPS and NAS both result in a very similar usefulness measurement. The difference is that

- with IPS you are prevented from making a piece of information useful more quickly as a consequence of multiple sub solutions existing in a solution.
- and with NAS precisely this "becoming useful more quickly" will be emphasized.

4 A FUNCTION FOR CALCULATING RELEVANCE

The relevance of information is calculated as a function of its age and usefulness, whereby the age counteracts the usefulness.

In this section the effects on relevance of the "becoming useful" and "aging" processes are separately considered and corresponding functions are stated. This leads us on to a formula for the calculation of relevance which combines both aspects.

4.1 Function for the aging of knowledge

We will now introduce the function forget($a_{i,tc}$) which we propose for technical domains. As a starting point we considered the power law of forgetting (see Introduction). The reason for this is that in technical domains similar characteristics are desirable for the development of relevance dependent on the aging of knowledge as in the human brain:

- New knowledge has maximum relevance and is much more relevant than old knowledge.
- Knowledge loses its relevance faster in the beginning. For example after 10 years one week either way will no longer have a great effect on the relevance of technical knowledge.
- The relevance only approaches zero whereas knowledge is never really irrelevant: the access just lasts longer. Real forgetting, i.e. the irreversible erasure of information is not desirable at first.

To make matters simpler the following assumptions were made compared with the function of the power law of forgetting.

- Reduction in the range of values to [0, 1]: The value for relevance should begin at 100% and approach 0%.
- A reciprocal function shifted to the left by 1 describes the desired effects just as well, such as an exponential function, is however more efficient for computers to calculate.

The first assumption is sensible because many parameters of the Cognitive Psychology are not available in the moment of storing the knowledge. For example in [15] it is described how the start relevance depends on the estimation of the importance of the information. We must assume however that all information in a knowledge base is from the same importance because a computer cannot make "emotional" assessments. The aforementioned "desired" characteristics remain, though.

The following definition describes the forget-function which we applied:

Definition 1:
The process of gradual forgetting information i in a knowledge base in technical domains can be described by the following function:

$$forget(a_{i,tc}): rel_{i,a,tc} = \frac{1}{a_{i,tc} + 1} \quad (3)$$

With

$a_{i,tc} \in [0,\infty[$ age of the information i in the task class tc (4)

$rel_{i,a,tc} \in]0,1]$ the part of relevance that is based on age for the task class tc (5)

Note: As time can be different for different task classes (e.g. when using NOR, number of runs, as the measurement for time) age can be dependent on the task classes. In the above definition this is indicated by the index tc.

4.2 Function for "knowledge training"

The Exponential Learning Function shows characteristics which seem to be adequate in technical domains:

- If information is used often, it seems to be important. The access time should be reduced, that's to say the relevance should be increased.
- The first accesses make information relevant more quickly than later accesses. If for example the same component is used for 10000 configuration processes, ten further accesses no longer have particular importance. Experts have thus learned this component is useful and will try it first.
- Relevance approaches any or the maximum relevance.

We reduce the range of values again to [0, 1] and use the reciprocal function shifted left by one as before in the forget function.

The following definition describes the train function we have used:

Definition 2:
The effect of the "training" of an information i in a knowledge base in technical domains can be described by the following function:

$$train(u_{i,tc}): rel_{i,u,tc} = 1 - \frac{1}{u_{i,tc} + 1} \quad (6)$$

With

$u_{i,tc} \in [0,\infty[$ Usefulness of information i for the task class tc (7)

$rel_{i,u,tc} \in [0,1[$ part of relevance that is based on usefulness for the task class tc (8)

4.3 Relevance of knowledge

In equation (1) the relevance function for an information i results from the addition of the forget and the train function. With equations (3) and (6), the following definition results

Definition 3:
The relevance of information in a knowledge base is described by the following function:

$$relevance(a_i t_c, u_{i,tc}) = c \cdot \frac{1}{m \cdot a_{i,tc} + 1} + (1\text{-}c) \cdot \left(1 - \frac{1}{u_{i,tc} + 1}\right) \in [0, 1] \quad (9)$$

With

$a_{i,tc} \in [0, \infty[$ Age of information i for a given task class tc (10)

$u_{i,tc} \in [0, \infty[$ Usefulness of information i for a given task class tc (11)

$c \in [0, 1]$ Constant for adjusting the weighting of usefulness and age (12)

m Constant for adjusting the measurement of usefulness and the measurement of time used (13)

The values can be interpreted as follows:

Table 1. interpretation of the relevance values.

0% to c	"normal" values for relevance (the range of values of the relevance function is almost always in this range).
c	Start value of relevance for new knowledge and threshold value for old but often useful knowledge.
c to 100%	Above-average relevance, which can only be achieved for a short period of time, if new knowledge is frequently useful right at the beginning.

As already mentioned, the constants c and m serve to weight the forget and train functions relative to each other. This is necessary for the following:

- The time and relevance measurements used must be adjusted to each other
- Depending on the domain age and usefulness can be varyingly important for the solution process

The constant c often can be set to 50% (c= 0.5) to get the same weighting for forget and train.

The following figure shows the relevance function from equation (9) with constants c=0.5 and m=1

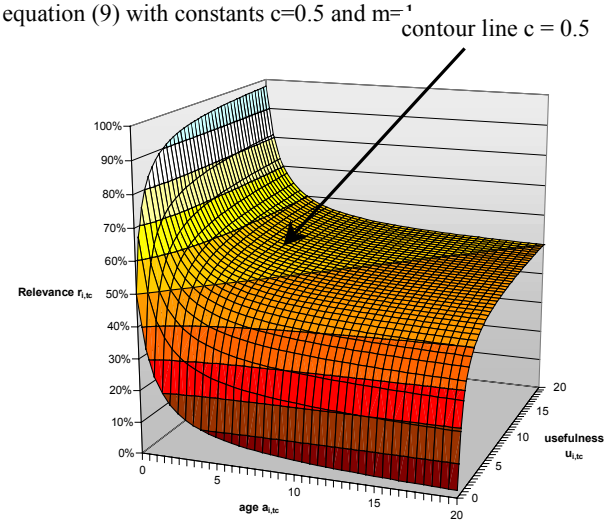


Figure 2. Relevance function with c=0.5 and m=1

As the values for usefulness and age increase to the same extent the relevance stays at c=50% (see Contour line c=0.5 in the figure). If the usefulness grows faster, relevance of over c and up to 100% ("behind" and "left" of the contour line c) can be achieved. Correspondingly the relevance slowly approaches 0% when usefulness grows slower in relation to age.

5 EXAMPLE OF APPLICATION

Before starting a configuration with RKF the objective specifications have to be defined. This happens by selecting a root (possibly also the root of a sub-tree) in the compositional hierarchy and a set of requirements which should be met by the configuration: e.g. functionality, components, parameters or

properties which a configuration should definitely have (e.g. colour "red").

Besides the aforementioned requirements, optimization criteria are typically also part of the objective specification. For example it would be desirable to configure the cheapest, most lightweight or fastest system possible. With RKF the optimization criteria are implicated by the selected task class which is also a component of the objective specification.

Starting from the selected root node of the objective specification it is attempted to recursively define all sub-components from the compositional hierarchy (and in other ways connected components). To this each component is specialized by means of the taxonomic hierarchy until a suitable and constructable part (leaf concept that can be instantiated) has been identified. For every component its parameters also have to be defined. RKF supports a depth first search in the taxonomic hierarchy whereby such leaf concepts are preferred which have a high relevance with respect to the selected task class. The determination of the range of values in the compositional hierarchy i.e. how often a component should be used as a sub-component can be supported by the relevance of specific values with RKF.

After some or even all configuration steps i.e. after every selection and setting of parameters of a (sub-) component, the consistency of the (sub) system must be guaranteed. If a conflict has occurred, it must be resolved e.g. by rejecting a selected component or a set parameter (backtracking). The actual configuration methods differ mostly in how they select components and resolve conflicts (sequence of selecting components, calculation algorithms and heuristics).

The provisional result is a parts list with a structure corresponding to the compositional hierarchy which describes all components together with their parameters, that belong to the configuration (solution). This solution is assessed.

Now the learning phase follows: The usefulness of all included specializations in the taxonomic hierarchy is increased corresponding to the solution's assessment. Equally the usefulness of specific values for the parameters of the components and the compositional relations are increased.

In the case that the configuration found is still not good enough relating to the above assessment, the configuration can be repeated in an optimization loop.

Our first configuration attempt with RKF has been carried out in a relatively simple domain: A PC needs to be configured using RKF from its individual components such as the drives, mainboard, memory etc. To this a depth first search and chronological backtracking has been implemented which is controlled by relevance. Figure 3 shows the compositional hierarchy of this domain.

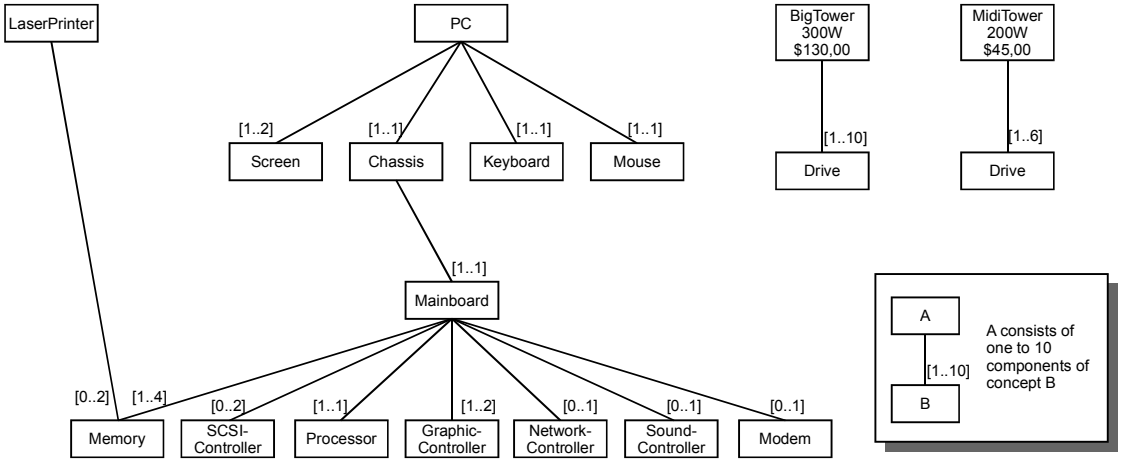


Figure 3: compositional hierarchy of the domain in our example

As a task specification one of these concepts are to be chosen. As a rule this is the concept "PC". Examples of task classes are "Home-PC", CAD-PC" and "Server-PC".

Figure 4 shows part of the taxonomy of the example domain. The above concepts are ancestor nodes of those concepts connected with lines below. For better clarity the alternatives of leaf concepts have been represented as a list one below the other (without further lines).

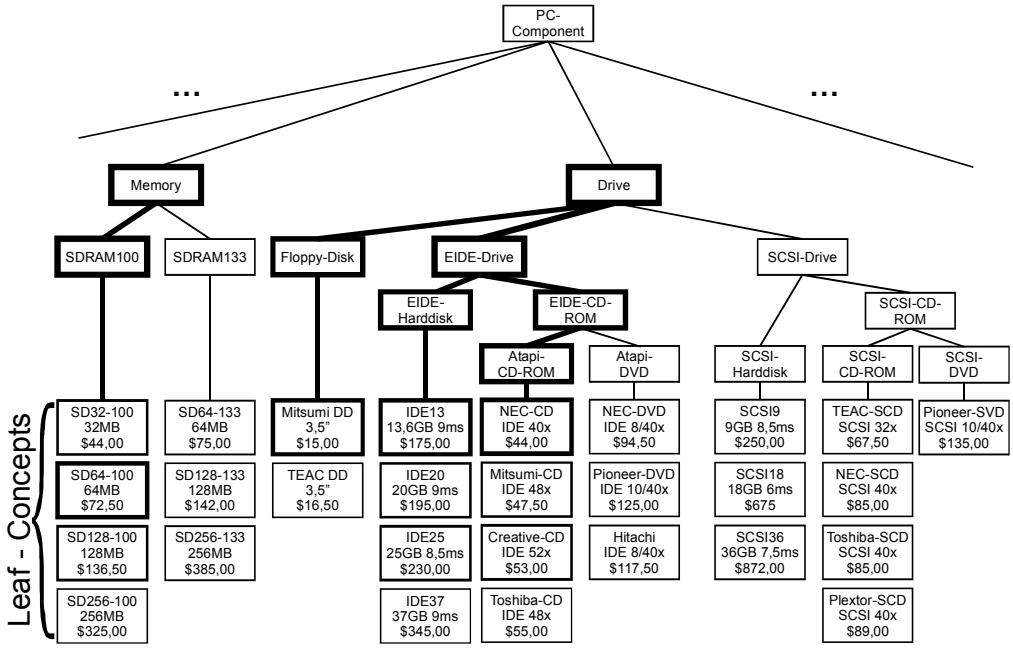


Figure 4. part of the taxonomy of the PC-domain. The thickness of the lines indicate the relevance for the task class "Home-PC"

The thickness of the lines indicate the relevance for the task class "Home PC" i.e. this form of PC most often contains a SD64-100 memory module, a Mitsumi DD-Disc drive, an IDE13- hard drive and a NEC-CD drive. The figure also shows how RKF supports the depth first search in this hierarchy: Paths of high relevance develop because whenever a daughter concept was useful, the respective father concept was also useful. The concept "PC-Component" has very minor relevance because it is not found in the compositional hierarchy. Instead of this the search always begins with a more specific concept (in the "drive" and "memory" examples). In other task classes of the same domain, the relevance can be completely different. For example the relevance of the SCSI hard discs for the "Server PC" task class is much higher than that of the EIDE hard disks.

The flexibility of RKF turns up as soon as a new component appears. Because of its low age it has a high relevance at the beginning despite its small usefulness. In the following example two new hard disk models (IDE22 and IDE27) are added to the taxonomy of the knowledge base.

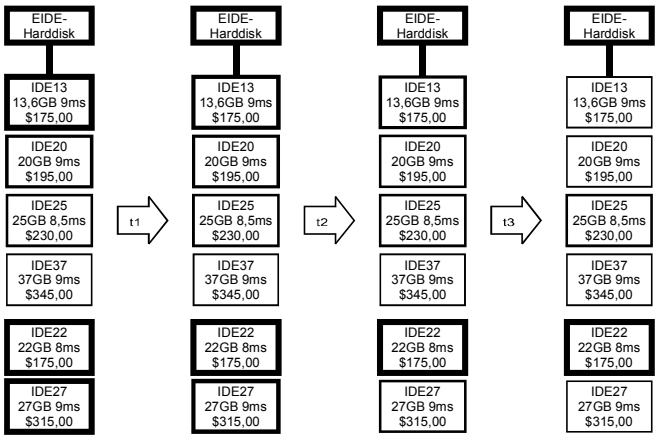


Figure 5. alteration of relevance over time for the task class "Home PC" after adding two new models of hard disks

The thickness of the lines in Figure 5 again represents the relevance of the individual concepts. The two new hard disks have a high relevance at the start because of their low age. Over the time period t1 to t3, the hard disk IDE 22 proves its worth for the "Home PC" task class and bit by bit replaces the hard disk IDE13 which until now was the most relevant, because of its aging without new usefulness. The second new hard disk does not prove its worth and loses its relevance quickly because of aging.

Incidentally the relevance of the father concept "EIDE hard disk" does not change if new concepts are added. That means that for the "Home PC" task class no "relevance trail" will develop for the SCSI hard disks, only due to the fact that a new disk has been added there. RKF reacts conservatively here: It is very unlikely that

RKF would find the new SCSI-hard disk without other heuristics (e.g. user-interaction). On the other hand RKF is flexible enough to adapt to market trends: If it suddenly became "modern" to have SCSI drives within the task class "Home PC", the relevance of these concepts would increase, because of the new demand and finally because of good overall rating. The relevance of the EIDE drives would become less because of aging.

6 SUMMARY AND OUTLOOK

The development of Relevant Knowledge First (RKF) was influenced by the effects of cognitive psychology. First of all age and usefulness were defined for use in RKF, and a function has been specified, which calculates the relevance of information from both these values. The specific characteristic of this method is that knowledge can not only be learnt, but also be "forgotten" without deleting information. This is always an advantage, if the information in a knowledge base is subjected to change. Knowledge bases in technical domains are usually dynamic due to innovations. As an example, the configuration in a PC-domain was presented using RKF. In this example first positive results of this method could ascertained.

Despite the positive findings we have already made using RKF the following questions need to be discussed:

One significant problem is the initialization of the usefulness for the different task classes: Can the initialization be performed automatically? I would say it depends on the way the assessment of the solutions works: If an assessment could be found, that can be calculated without any user interaction this could be achieved. In this case we seem to find a machine that can tell us the future. Because no such oracle has been found yet it might help to have a look at past configuration results. For example PCs sold before the configuration system for PCs was installed.

On the other hand this reminds us of neural networks during their training phase. One could say that the learning part of RKF has some similarity, whereas RKF additionally takes aging particularly into account.

Another issue to discuss, is the definition of the relevance function given in this article. Of course this is only one possibility and yet we have many other ideas which differ slightly. For example for some domains it seems sensible to stress *when* information was successfully used *recently*.

I'm looking forward to the discussion on these issues during the workshop.

REFERENCES

[1] John R. Anderson: *Kognitive Psychologie.* Deutsche Übersetzung von Joachim Grabowski und Ralf Graf. 2. Auflage. Spektrum Akademischer Verlag Gmbh Heidelberg, Berlin, Oxford, 1996, ISBN 3-86025-354-9 Pb. ISBN 3-8274-0085-6 brosch., chapter 6 und 7

[2] Roman Cunis, Andreas Günter, Helmut Strecker*: Das Plakon-Buch. Ein Expertensystemkern für Planungs- und Konfigurierungsaufgaben in technischen Domänen*, Springer-Verlag, Berlin Heidelberg, 1991, ISBN 0-387-53683-3

[4] Marco Dorigo, Vittorio Maniezzo and Alberto Colorini: *The Ant System: Optimization by a colony of cooperating agents*, IEEE Transactions on Systems, Man and Cybernetics-Part B, Vol. 26, No.1, 1996, pp. 1-13

[5] Andreas Günter (ed.*)*: *Wissensbasiertes Konfigurieren. Ergebnisse aus dem Projekt PROKON.* Infix-Verlag, Sankt Augustin, 1995, ISBN 3-929037-96-3

[6] Andreas Günter, Christian Kühn*: Knowledge-Based Configuration – Survey and Future Directions – in Knowledge based systems: survey and future directions*; proceedings / XPS-99, Würzburg, Springer Verlag Berlin, Heidelberg 1999, ISBN 3-540-65658-8, pp. 47-66

[7] Andreas Günter, Ingo Kreuz, Christian Kühn*: Kommerzielle Software-Werkzeuge für die Konfigurierung von technischen Systemen* in *KI – Künstliche Intelligenz* Heft 3/1999, arenDTaP Desktop Publishing Agentur, Verlags- und Vertriebs GmbH, Bremen, 1999, ISSN 0933-1875, pp. 61-65

[8] Albert Haag: *Sales Configuration in Business Processes* in *IEEE Intelligent Systems* Jul/Aug 98, 1998, pp. 78-85

[9] Heinrich, Jüngst: *The Resource-Based Paradigm: Configuring Technical Systems from Modular Components* in AAAI-96 Fall Sympos. Series: Configuration. MIT, Cambridge, MA, November 9-11, 1996, pp. 19-27

[10] Ingo Kruez, Thomas Forchert, Dieter Roller: *ICON. Intelligent Configuring System* in Dieter Roller (ed.) *Proceedings of the 31th ISATA*, Volume "Automotive Electronics and New Products", Düsseldorf Trade Fair, Croydon, England 1998, ISBN 0 9532576 5 7, pp. 219-226

[11] Ingo Kreuz, Ulrike Bremer: *Exact Configuration Onboard. Onboard Documentation of Electrical and Electronical Systems consisting of ECUs, Data Buses and Software*, ERA conference 1999, Coventry, UK, 1999, ISBN 0-700806-95-4 pp. 5.2.1-5.2.8

[12] Ingo Kreuz, Dieter Roller: *Knowledge Growing Old in Reconfiguration Context*, in Boi Faltings, Eugen C. Freuder, Gerhard Friedrich, Alexander Felfernig *Configuration, Papers from the AAAI Workshop*, Technical Report WS-99-05, Orlando, 1999, ISBN 1-57735-089-8, pp. 54-59

[13] J. McDermott: *R1: A Rule-based Configurer of Computer Systems* in Artificial Intelligence 19(1), 1982, pp. 39-88

[14] Elaine Rich, Kevin Knight: *Artificial Intellligence, Second Edition.* McGraw-Hill Book Co, Singapore, 1991, ISBN 0-07-100894-2.

[15] David Waltz: *The Importance of Importance* in *AImagazine*, Volume 20, No. 3, Fall 1999, American Association for Artificial Intelligence (AAAI), Menlo Park, CA, USA, 1999, ISSN 0738-4602, pp. 18-35

Modeling Structure and Behavior for Knowledge-Based Software Configuration

Christian Kühn*

Abstract. There are several approaches to knowledge-based configuration, which are successfully applied in technical domains. Nevertheless they appear not to be sufficient for configuring software-based systems, as they primarily proceed in a structure-based manner, but do not take the system's behavior into account. In this context, the configuration of software-based systems does not mean the programming of software, but the composition of existing software modules into an individual software variant. Our approach uses state-based behavior descriptions for domain objects, which can be directly used for making decisions during the configuration process. Our goal is to apply this approach for a knowledge-based configuration of future software-based vehicle electronic systems, which implement customer-individual vehicle functions.

1 INTRODUCTION

The configuration of complex software-based systems which can carry out a high number of variants can be a difficult and time-consuming task. In this context, configuration does not mean the programming of software, but the composition of existing components into an individual software variant. In the area of Artificial Intelligence, various methods for configuration have been developed that could already be successfully applied, especially for technical systems. The majority of these methods are structure-based, i.e. the configuration process as well as the specification of the configuration objectives are based on the structure of the system to be configured. For the configuration of software-based systems these structure-based approaches seem to be suitable, but not sufficient. In particular, the consideration of the software system's behavior appears to be very significant. What software *does* can be more important than how it is structured. There are existing approaches, which in principle take behavior during configuration into account. However the application of behavioral knowledge is rather limited to the evaluation of solutions or partial solutions. An extension of modeling techniques is needed, to adequately enable model software behavior, which can be utilized for configuration.

Once again it must be reiterated that our approach is not aimed at the process of individual software development or programming. It is intended for building up a high number of individual software systems which are based on the same set of basic components or modules. At the time of configuration the process of module development has already been finished. Furthermore our method is intended rather to configure combined hardware/software systems (embedded systems) than pure software systems. Nevertheless we will focus on the software aspect in the following. Software can be considered as a domain, while software modules are understood as domain objects and will be referred to as concepts in the knowledge base (software module concepts).

In the following section (sec. 2) a short introduction into current knowledge-based configuration approaches will be given with a focus on structure-based and behavior-based techniques. The limits of these approaches will be pointed out. Section 3 describes our proposed extension of current structure-based configuration knowledge modeling by behavior models, which serve as a basis for the configuration process (sec. 4). In section 5 the configuration of software-based vehicle electronic systems will be illustrated as an example of application. This paper ends with a summary and brief outlook (sec. 6).

2 APPROACHES TO KNOWLEDGE-BASED CONFIGURATION AND THEIR LIMITS FOR SOFTWARE CONFIGURATION

Knowledge-based configuration belongs to the class of synthesis tasks. For both the representation of knowledge and the reasoning process different methodologies have been developed. The following examples are the most relevant (see [10]):

- *Structure-based approach:*
  In the structure-based approach a compositional, hierarchical structure of the domain objects serves as a guideline for the control of problem solution.
- *Constraint-based approach:*
  Representing restrictions between objects or their properties, resp. by constraints and evaluating these by constraint propagation. The constraint-based approach is not in competition with the structure-based approach but is frequently combined with it.
- *Resource-based approach:*
  Resource-based configuration is based on the following principle, that interfaces between components are specified by the exchanged resources. Components make a number of

* DaimlerChrysler AG, Research and Technology, HPC T721, D-70546 Stuttgart, Germany, e-mail: christian.kuehn@daimlerchrysler.com

resources available and also consume resources themselves. A task specification exists in form of required resources.

- *Case-based configuration:*
  Case-based problem-solving methods are therefore identified, so that knowledge concerning already-solved tasks is saved and is used for the solution of new tasks. The reason for this is that similar tasks lead to similar solutions.

Next to these approaches for knowledge-based configuration there are several other techniques, like rule-based techniques and especially techniques that combine different methods, e.g. using simulation, optimization or spatial reasoning throughout the configuration process. Some of these approaches which look at behavior, will be dealt with in the following subsection (2.2). We will now consider the structure-based approach for software configuration.

2.1 Structure-based configuration

Under the term configuration, the step by step assembly and parameter setting of objects from an application domain to a problem solution or configuration is understood, where certain restrictions and given task objectives should be fulfilled (see [10], [17]).

A configuration problem comprises the following components (see [10]):

- A set of objects in the application domain and their properties (parameters).
- A set of relations between the domain objects, while taxonomic and compositional relations are of particular importance for configuration.
- A task specification (configuration objectives) which specifies the demands a created configuration must fulfill.
- Control knowledge about the configuration process.

The process of configuration consists of a sequence of configuration steps. As the domain model describes the set of all possible solutions (configurations), each configuration step can limit this set until it is reduced to a final solution. In order to distinguish between the knowledge representation level and the solution level, the objects on the knowledge representation level will be called domain objects or concepts, and the solution elements will be called instances of these concepts. Possible configuration steps are specialization (refining an instance to a more specific concept), decomposition (top-down instantiation of an aggregate's components), integration (bottom-up including an instance into an aggregate), and parameterization (determining an instance's property value). Examples of configuration systems based on these types of configuration steps are PLAKON [4], KONWERK [6], [8], and EngCon [1].

An overview of the technologies, applications and systems is given in the above mentioned literature.

2.2 Utilizing behavior models for configuration

Some new approaches take the dynamic system's behavior over time into consideration; this concerns principally the integration of simulation techniques in knowledge-based configuration (see e.g. [3], [9], [15], [16]). The aim on one hand is to reduce partial solutions and calculate values through simulation. On the other hand simulation can be used for testing the behavior of a fully or partly configured system and therefore facilitates the verification with the subsequent evaluation of (partial) configurations.

For verification and evaluation of a configuration or partial configurations the results of other (heuristic) problem solving methods are verified through simulation. This can be integrated within a configuration system in the following ways (see [9]):

- Mapping of the knowledge base and the partial configuration onto a suitable model for simulation,
- Simulation of this model and
- The evaluation of the simulation through the configuration system.

There are simulation procedures which are integrated into configuration both in the area of quantitative continuous simulation (e.g. [9], [15], [16]) and in the area of condition-based, qualitative simulation (e.g. [3]).

2.3 Limits of the current techniques for software configuration

The above described structure-based techniques seem to be suitably applicable to configure complex software-based systems, as these systems are based on a module structure. Although these techniques provide a good basis for configuring software systems, they are not yet sufficient as most of them are confined to the structural construction of systems without taking the systems' behavior into consideration. The reactive system behavior can become very complex, especially for software-based systems.

An area of application similar to configuration, is the planning that also belongs to the class of synthesis tasks. For planning, time and temporal aspects play a central role (see [2]). However in planning, the target object is a sequence of operations, which convert a starting state into a target state. In contrast to configuration, here the organization of the different planning steps along a temporal axis is of great importance. However in configuration the aim is to compose a system (i.e. finding a structure), which in our context has a desired reactive behavior. Both tasks are similar, but the focus is different.

Often in the context of the "software" domain, a clear module structure which can be transferred to the knowledge base has already been worked out during the software development process. In contrast to this, specific knowledge about properties and behavior[1] of the software modules have to be supplemented.

[1] In this context behavioral knowledge does not mean the detailed behavior e.g. on code level, but abstract knowledge about the behavior that can be utilized for configuration.

Although on principle the above-mentioned approaches to the integration of simulation into the knowledge-based configuration take the behavior of a configuration system into account, they are not sufficient for the development of complex, software-based systems, since they are not designed for the high variance of potential system behavior. Instead of this, a modeling is needed which allows an adequate, abstract software modeling, which makes direct conclusions possible from the modeled behavior of software objects (meaning software modules as domain objects).

Therefore we propose an approach which forms the basis of a combination of structural configuration knowledge and behavior-based configuration knowledge. (The structural knowledge means domain objects with properties, which are organized in taxonomic and compositional hierarchies. The behavior-based knowledge being statecharts allocated to the domain objects.)

3 EXTENDING CONCEPT HIERARCHIES BY ABSTRACT SOFTWARE BEHAVIOR MODELS

The aim is to model knowledge of the usable software modules for configuration, so that reasoning from this knowledge is possible. This is described in the demands on modeling techniques and problem solving methods for software configuration in [14]. A declarative, generic modeling of software modules as abstract concepts is proposed. These concepts (as well as the other components), are arranged in a concept hierarchy and are provided with attributes. For example, several concepts of the same software module can be specified, but with different ranges of detail. Likewise, the same concept can be described through its generic representation of several software modules, when these modules are represented by an abstract superconcept because of common properties.

Of course, with a description of a software module by a domain object (a so-called concept), there are numerous properties that can be specified, e.g. development data (version, authors, short documentation), hardware allocation (suitable processors, required memory of the module (ROM), required memory during runtime (RAM), applied periphery components, properties for application (input/output interfaces). We want to extend these "simple" concept properties for software concepts by behavior models as complex concept properties.

There are different ways of describing behavior. A simple way to describe the behavior of a software module, is by using the explicit allocation of one or more buzzwords which describe the eligibility and possible applications of the module, e.g. *light activation* or *light dimming*.

The functionality of a module can be represented in more depth by state models, e.g. finite state machines or Petri nets, just as with rules. There are different possibilities. It is not intended to model the behavior as detailed as on code level, but to give an abstract description of the behavior. In the following, statecharts [13] are used as a description method for the behavior of modules, which we will go into in more depth in the subsequent sections.

3.1 Assigning state behavior to domain objects

Statecharts are an extensive method, their functionality is described in detail in [13]. In contrast to state machines, statecharts allow (among others):

- Modularization (that means that single states can be refined to statecharts themselves),
- Parallelism (that means that several states can be active at the same time, i.e. states can be orthogonal) and
- Real-time conditions.

Especially in the context of embedded systems, statecharts are often used for system (and software) specification. In this way statecharts are frequently used for the generation of executable code (e.g. C or C++) for embedded systems (see [5], [11], [13]). In contrast to this, in our approach statecharts serve exclusively for the abstract description of (software) behavior as a starting point for selecting existing software modules and appropriately combining them. The aim is not to generate code (see above).

The same state-based behavior, which is described in our approach by statecharts, can equally be described with other techniques, e.g. with finite state machines, regular expressions, or the explicit set of all traces or sequences of state changes over time (as can be seen in figure 2). While the modeling by the user or knowledge engineer is generally simpler and more adequate on statechart level[2], the internal handling of behavior is however simpler using one of the other techniques – e.g. finite state machines – as operations such as comparison or specialization are easier to perform. Thus each modeled statechart can be converted into an internal representation before it is used in the configuration process.

In the approach described here, only parts of the general statechart concepts are utilized. It concerns state transition diagrams, whose transitions are labeled with triggering conditions. Corresponding to this a state can have transitions to several target states, whereas transitions are flagged with different conditions. Conditions can base on external events or on internal events (for example entry into or exit from states). In addition such events can be combined by Boolean operators with further conditions, e.g. with information about the activity or inactivity of other states. The usable operators for describing conditions (in addition to the Boolean operators *and*, *or*, *not*) are listed in table 1.

[2] In particular, statecharts are fundamentally concerned with finite specifications, whilst traces can be infinite. In comparison to finite state machines, statecharts are much more compact. Regular expressions are too difficult to handle for the user.

operator	description
in(instance, state)	The specified state has to be active.
some-in(set of instances, state)	The specified state has to be active for one of the instances.
all-in(set of instances, state)	The specified state has to be active for all of the instances.
entry(instance, state)	Transition to the specified state takes place.
some-entry(set of instances, state)	Transition to the specified state takes place for one of the instances.
all-entry(set of instances, state)	One instance passes into the specified state, afterwards all instances of the set are in this state.
exit(instance, state)	Transition from the specified state takes place.
some-exit(set of instances, state)	Transition from the specified state takes place for one of the instances.
all-exit(set of instances, state)	One instance exits the specified state, afterwards no instance of the set is in this state.
timeout([event], duration)	Transition is triggered by a real-time condition: after the specified event has happened (or after entering the current state, if no event is given), and the given duration of time has past.
extern	Transition is triggered external.

Table 1. Operators for specifying conditions for state transitions

Statecharts are assigned to the individual domain objects (in this case software modules) as properties of these components. For this, state variables with a set of permissible states (which are alternatively, i.e. exclusively, active) can be assigned to the domain objects. In our approach the statecharts of different concepts can be based on the same states, i.e. a domain object's statechart can also use other domain objects' states, or at least reference other domain objects' states in its conditions for transitions. To do this a domain pattern can be specified which describes the referenced states.

An important point about configuration is that during the configuration process it is not necessarily definite, which domain objects are relevant for the solution, i.e. which will be instantiated as a part of the solution (consider the distinction of concepts and instances in chapter 2). To access any set of instances' states in the statecharts we introduce predicate logical condition operators for transitions (some-in, all-in, some-entry, all-entry, some-exit, all-exit). Figure 1 shows a state transition taking place, if at least one vehicle door is opened or all doors are closed, as an example.

A classification in a concept hierarchy requires that objects can be specialized, decomposed, and parameterized together with their behavior (see above). Whereas parameterizing the behavior models does not play a significant role in our approach, we are focusing on specializing modeled behavior (in the taxonomical hierarchy) and decomposing modeled behavior (in compositional hierarchies) in the following.

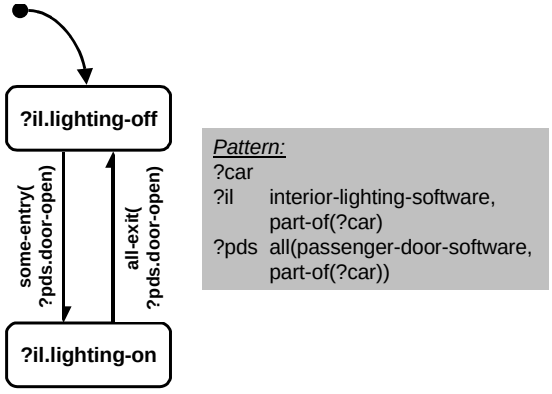


Figure 1. Example of a statechart

3.2 Behavior models in the taxonomical structure

Let us consider the specialization of concept C to concept C'. Each property of C' – also the behavior – must potentially be a property of C. For behavior that means that the set of C' traces must be a subset of the C traces (this also corresponds to the comprehension of behavioral inheritance in [11] and [12]). The superconcept's more extensive set of traces means a greater scope of behavior alternatives, whereas specialized objects have less alternatives for their behavior as they are described more specifically. Figure 2 shows the specialization of behavior of an example concept. For building up the taxonomical hierarchy it is enough to model the behavior of the domain objects on the lowest specialization levels. All higher levels can be automatically produced, for example by unifying the behavior traces (or unifying the finite state machines corresponding to the statecharts, resp., depending on what internal representation is used; see above).

3.3 Behavior models in the compositional structure

As an aggregate A is composed from components (e.g. from C1 and C2), the corresponding statechart can also be composed from the components' statecharts. Considering the traces we can build combinations of the states in the aggregate's components up to new states in the aggregate's traces, as can be seen in figure 3. Although in the example the statechart corresponding to A consists of two parts graphically, it is a closed model whose parts are connected with each other by the entry conditions.

Decomposing an object into its components permits a high number of options in general, as not all potential components of an object (or a module resp.) need be choosen (and instantiated) in a given configuration. A decomposition is only permissible if the restrictions between the partial components' statecharts, which are given by domain patterns, are fulfilled. In the example (fig. 3) the pattern variables *?x* and *?y* of the component C2 can be mapped onto the states *c* and *a* of the component C1.

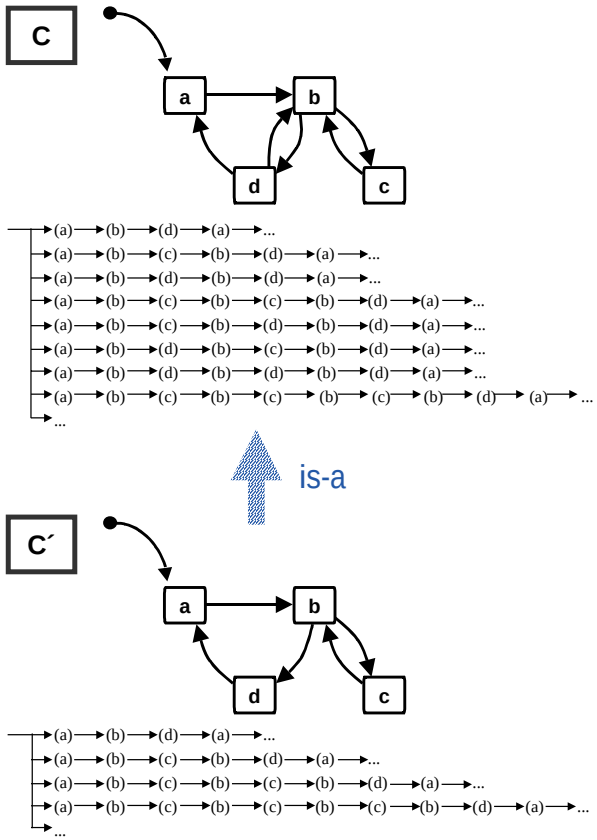


Figure 2. Specialization of a concept's behavior, described by statecharts and state traces

In the following section it will be described how the behavior models can be used to draw conclusions during the configuration process.

4 CONFIGURATION PROCESS USING BEHAVIOR KNOWLEDGE

Both the structural knowledge (arrangement of domain objects in a taxonomic hierarchy with relationships and parameters) and the behavior knowledge (state-based behavioral description as statecharts or any other internal representation) can serve as a starting point for configuration decisions. In this context, behavior-based decision-making should not compete with structure-based decision-making, rather they should complement one another.

Also the specification of a configuration goal can comprise of both structural and behavioral requirements. This means that when the configuration objectives are entered, the target object and required properties[3] can be specified just as well as particular requirements for the behavior of the system to be configured, which in our approach can be specified by (incomplete) statecharts. The total behavior should not have to be stipulated by the user (just as little as the whole structure of the system), this is "configured" in the solution process.

Whilst we will not be going further into how knowledge structure is utilized for configuration decisions (for this see e.g. [7], [10]), it will be described how the behavior knowledge can be used to perform the four types of configuration steps given in section 2 (specialization, decomposition, integration and parameterization). Correspondingly it could be possible to extend the control of the PLAKON, KONWERK or EngCon systems (see [4], [6], [1]).

Specializing an instance. The specialization of a software module instance means to transmute this instance into a more specific module, i.e. (among other effects) to reduce the potential value ranges of its properties. For the behavior this means that the set of possible traces will be reduced to a subset (see above). In this way, the behavior model description can be transmuted – by transmuting the instance into a more specific subinstance – into a specialized behavior, which is suitable for the behavior demanded in the task specification. To ensure this, the specialized behavior has to be checked against the statechart in the task specification.

Decomposing an instance. While a module instance is decomposed, its accessory submodules could be instantiated and become elements of the current solution. As well as the modules being decomposed, the behavior models belonging to these modules are also decomposed. Concomitant to the decomposition and instantiation of components, relating the participated statecharts to each other can be performed on the basis of bindings in the domain pattern. That is to say, variables in the domain pattern can be bound to the respective instance. This enables the accessing of respective instances in the statecharts' conditions.

Integrating an instance. Integrating a component into an aggregate means exchanging a generic subconcept in the aggregate by a specific instance of this concept. As said before the behavior of the instance might be more precise than the concept's behavior (as the set of traces might be reduced). It has to be guaranteed that this specialized behavior still suits the other subconcepts or instances of the aggregate. If this is not given, the configuration control will not permit this integration step. As with integration, an existing instance (reducing its concept's scope) is assigned to an aggregate, the integration can also be seen as a kind of specialization of the aggregate.

[3] See for example [18].

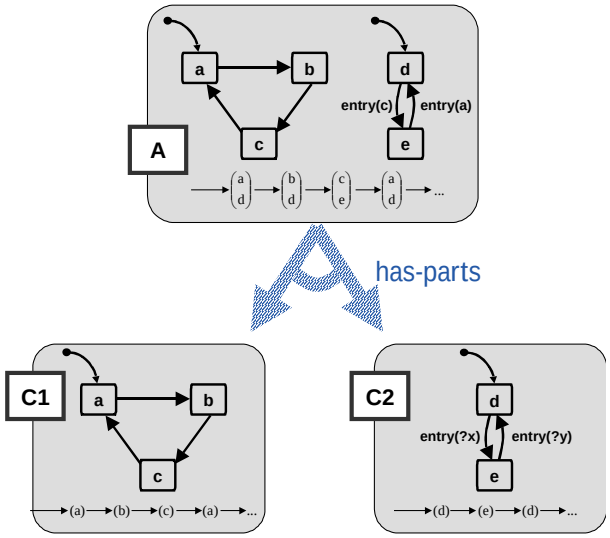


Figure 3. Consideration of behavior during aggregation of C1 and C2 to A

Parameterization of an instance. Fundamentally, parameters in the behavior description can be handled similarly to the remaining parameters (properties) of an instance, i.e. parameterization also takes place here by a reduction of the range of values for the behavior parameters.

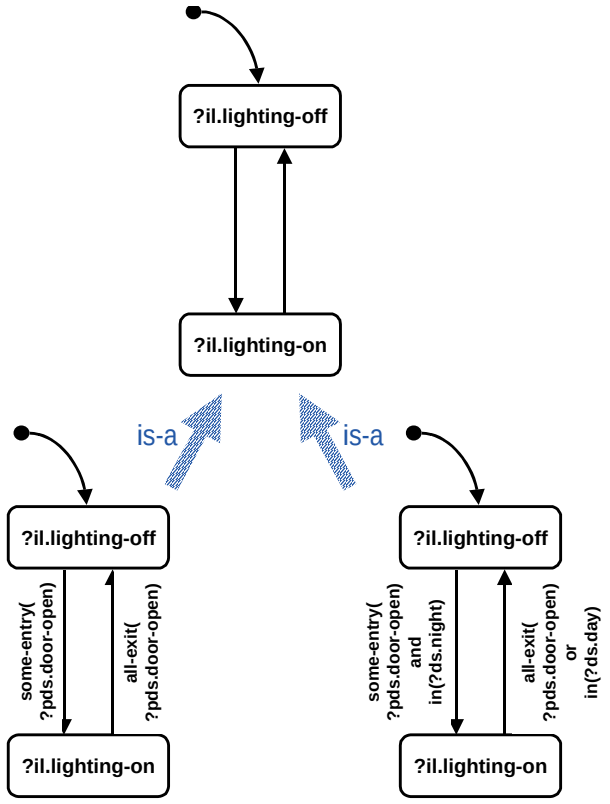


Figure 4. State descriptions of the software module *interior lighting* and its specializations

The configuration process, consisting of specialization, decomposition, integration and parameterization steps – as well as further techniques such as constraint propagation, which are not described further here – should result in a set of instances, where each can have behavior. This behavior has to be unequivocal for the configuration solution. As the behavior of all instances are based on the same set of states, all behavior models can be seen as one behavior model. This total behavior must fulfill all behavioral requirements given in the task specification by the user.

5 APPLICATION EXAMPLE: CONFIGURATION OF SOFTWARE-BASED VEHICLE ELECTRONIC SYSTEMS

In the field of current vehicle electronics, the trend of increasing the functionality using software can be noted. The introduction of software provides the possibility of using universal control units in the future, instead of specialized control systems for different functions (e.g. control systems for engines, gears, suspension, etc.). These can

- be programmed easily.
- be modified later.
- carry out several sub functions at the same time.

By giving different vehicles individual software configurations, higher diversity and more customer-appropriate vehicle construction can be achieved. This trend leads to increased demands on the development of the electronic system within the vehicle. On one hand there will be an immense variety of possible variants, whereas on the other, the variants will be subject to increasing change through the simple realization of software updates.

The knowledge-based software configuration approach based on the above described software modeling, gives the ability to support the creation of the high number of vehicle software variants in vehicle development as well as in sales. In sales, the knowledge-based configuration can help to realize individual customer-desired functions, by using an appropriate software configuration for each car. To make this possible, such a configuration has to be based exclusively on hardware and software components which are permitted for the respective vehicle series. Particularly in sales, individual programming of control units performed by engineers and specialists should be avoided. The task rather concerns the composition of the vehicle software from single modules and their parameterization. An example for a behavior model of a software module *interior lighting*, which can be specialized to a *daylight-independent interior lighting* and a *daylight-dependent interior lighting*, is shown in figure 4.

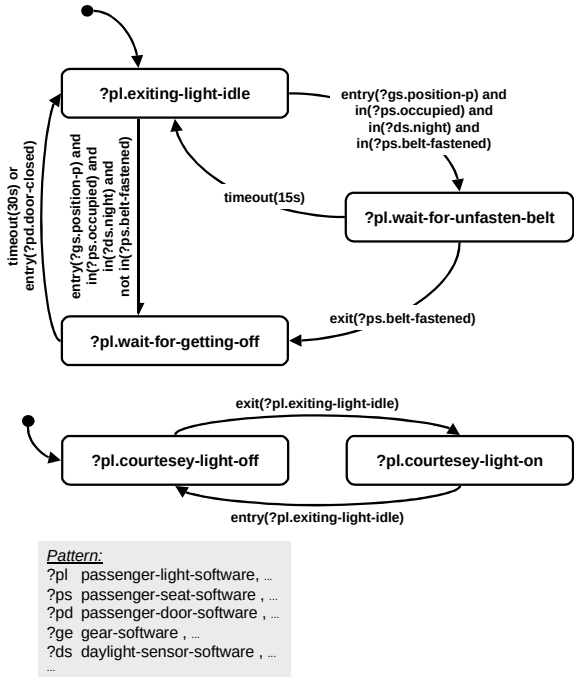


Figure 5. Statechart description of an individual customer demand for the courtesy light

A scenario by way of example: Let's say a customer (e.g. a taxi driver) wants his car to be equipped with a particular courtesy light, that helps the passenger to locate the seat belt lock and the handle. This should be achieved by the automatic illumination of the interior lighting on the passenger's side as soon as the driver switches the automatic gear stick to "P" (parking). If the passenger does not leave the car within a predefined period, the light should switch off automatically.

A sales employee could record such a customer demand by means of a (not necessarily complete) statechart, which could serve as a task specification for the configuration system (see figure 5). The configuration system's task is now to determine the software modules to be used, to parameterize them, to distribute them onto the existing control units and to determine their sequences. This includes the determination of the bus communication for each control unit.

6 SUMMARY AND OUTLOOK

Configuring complex software-based systems, which are able to carry out reactive behavior – e.g. the software-based electronic systems of vehicles in the future – can be a very intricate task. Configuration techniques that exceed current configuration approaches are required. The system behavior should especially be taken into account during configuration. We suggest a modeling approach for knowledge on software, which is based on a software module concept hierarchy, implying taxonomical, compositional and interface relationships. Beside other module properties, each module concept can be given a description of the module behavior on an abstract level, e.g. using statecharts. These behavior descriptions can be used for drawing configuration decisions such as specializing, decomposing, integrating or parameterization of module concepts. As the individual configuration steps can be based on this behavior knowledge, our approach extends to current structure-based configuration methodologies.

In opposition to simulation-based configuration approaches our method allows performing of the basic configuration steps (specializing, composing, integrating or parameterization) directly based on the behavior model, whereas when using simulation, configuration decisions first have to be driven (using a heuristics) and afterwards the results can be evaluated by simulation.

This paper is confined to the basic types of configuration steps. There are further techniques, especially constraint propagation, which promise to have favorable effects if they were also extended to use behavioral configuration knowledge. By way of contrast to the behavior descriptions given here, which are each assigned to the domain objects (and always assigned to *a particular* domain object), constraints represent restrictions between several domain objects. In this way they should also be able to describe *behavioral* restrictions between domain objects. A corresponding extension of the current constraint techniques for configuration, which so far disregard temporal (and therefore behavioral) dependencies is needed. This topic may be dealt with in the future.

For the future, the implementation of a prototypical configuration system is planned, which comprises the methods outlined in this article. It shall be used for configuring vehicle software systems to implement customer-individual vehicle functionality.

REFERENCES

[1] Arlt, V.; Günter, A.; Hollmann, O.; Wagner, T.; Hotz, L. *EngCon – Engineering & Configuration*. In Friedrich, G. ed. *Configuration – Papers from the AAAI Workshop (Orlando, Florida)*, Technical Report WS-99-05, AAAI Press, California, 1999

[2] Biundo, S.; Günter, A.; Hertzberg, J.; Schneeberger, J.; Tank, W. *Planen und Konfigurieren*. In Görz, G. ed. *Einführung in die künstliche Intelligenz*, 2nd edition, Addison-Wesley Publishing, Berlin, 1995

[3] Bradshaw, J.; Young, R. M. *Evaluating Design Using Knowledge of Purpose and Knowledge of Structure*. In IEEE Expert, Vol. 6 (2), pp. 33-40, 1991

[4] Cunis, R., Günter, A., Strecker, H. ed. *Das PLAKON Buch – Ein Expertensystemkern für Planungs- und Konfigurierungsaufgaben in technischen Domänen*, Springer, Berlin, 1991

[5] Douglass, B. P. *Real-Time UML – Developing Efficient Objects for Embedded Systems*, Addison Wesley, Reading, Massachusetts, 1998

[6] Günter, A. KONWERK – ein modulares Konfigurierungswerkzeug. In Maurer, F.; Richter, M. M. eds. *Expertensysteme 95*, Kaiserslautern, infix Verlag, pp. 1-18, 1995

[7] Günter, A. ed. *Wissensbasiertes Konfigurieren – Ergebnisse aus dem Projekt PROKON*. St. Augustin, Germany: infix Verlag, 1995

[8] Günter, A.; Hotz, L. *KONWERK – A Domain Independent Configuration Tool*. In Friedrich, G. ed. *Configuration – Papers from the AAAI Workshop (Orlando, Florida)*, Technical Report WS-99-05, AAAI Press, California, 1999

[9] Günter, A.; Kühn, C. *Einsatz der Simulation zur Unterstützung der Konfigurierung von technischen Systemen*. In Mertens, P.; Voss, H. eds. *Expertensysteme 97 – Beiträge zur 4. Deutschen Tagung Wissensbasierte Systeme (XPS-97)*, Bad Honnef am Rhein, infix Verlag, pp. 93-106, 1997

[10] Günter, A.; Kühn, C. *Knowledge-Based Configuration – Survey and Future Directions*. In Puppe, F. ed. *XPS-99: Knowledge Based Systems, Proceedings 5th Biannual German Conference on Knowledge Based Systems*, Springer Lecture Notes in Artificial Intelligence 1570, Germany, 1999

[11] Harel, D.; Gery, E. *Executable Object Modeling with Statecharts*. IEEE Computer, Vol. 30, No. 7 (July), pp. 31-42, 1997

[12] Harel, D.; Kupferman, O. *On the Inheritance of State-Based Object Behavior*. To appear, 2000

[13] Harel, D.; Politi, M. *Modeling Reactive Systems with Statecharts*. McGraw-Hill, 1998

[14] Kühn, C. *Requirements for Configuring Complex Software-Based Systems*. In Friedrich, G. ed. *Configuration – Papers from the AAAI Workshop (Orlando, Florida)*, Technical Report WS-99-05, AAAI Press, California, 1999

[15] Kühn, C.; Günter, A. *Combining Knowledge-Based Configuration and Simulation*. In Kleine Büning, H. ed. *Beiträge zum Workshop „Simulation in Wissensbasierten Systemen“ (SiWiS-98)*, report tr-ri-98-194, Universität-GH Paderborn, Germany, 1998

[16] Stein, B. *Functional Models in Configuration Systems*. Dissertation, University of Paderborn (GH), 1995

[17] Stumptner, M. *An overview of knowledge-based configuration*. AI Com 10 (2), pp. 111-126, 1997

[18] Thärigen, M. *Wissensbasierte Erfassung von Anforderungen*. In Günter, A. ed. *Wissensbasiertes Konfigurieren – Ergebnisse aus dem Projekt PROKON*, 89-96. St. Augustin, Germany: infix Verlag, 1995

Benefits and Problems of Using Cycle-Cutset Within Iterative Improvement Algorithms

HARALD MEYER AUF'M HOFE
GWI-SIEDA GmbH
Richard-Wagner-Str. 91, 67655 Kaiserslautern
Email: Harald.Meyer@gwi-ag.com

Abstract. Some experiments on randomly generated partial constraint satisfaction problems as well as an example from the domain of real world nurse rostering illustrate the advantage of the cycle-cutset method as a repair step in iterative search. These results motivate the integration of adopted algorithms on solving tree-structured constraint problems and the cycle-cutset method into modern constraint-based optimization with branch-and-bound and propagation of global constraints.

1 Motivation

The GWI Group is the number two provider of integrated software for hospitals in Germany. Process management in hospitals requires the generation of several schedules for instance on nurse rostering, transport of patients, and coordination of diagnosis processes. The availability of new therapies in combination with increasing cost pressure motivates further optimizations of these schedules by intelligent optimization systems. The GWI-SIEDA GmbH pioneered this approach by the development of a constraint-based nurse rostering system that is used in a growing number of hospitals [Meyer auf'm Hofe, 1997, Meyer auf'm Hofe, 2000, Meyer auf'm Hofe, 1999].

All these applications require the use of generic algorithms that work on declarative and easily maintainable problem representations since the exact scheduling problem differs typically from hospital to hospital. Thus, the above cited system uses very general methods: Extended *partial constraint satisfaction problems (PCSP)* [Freuder and Wallace, 1992] are used to represent the rostering problem. An iterative search where a branch-and-bound algorithm extended by constraint propagation conducts improvement steps is used to produce rosters. Soft constraints in combination with iterative search allows on-line modification of the problem specification. Additionally, iterative search converges quickly on rosters of sufficient quality.

Refer to Fig. 1 for a small and simple example. This figure presents a simplified roster. Each cell is labeled by a shift that is either a early-morning shift (MS), a late shift (LS), a night shift (NS), a longer day-turn (DT), or an idle shift (-). The corresponding constraint problem contains a constraint variable for each cell in the roster. The available types of shifts form the domain of these variables. Constraints concern either shift assignments within the same row of the roster, e.g. due to working time restrictions and compatibility of consecutive shifts. Or constraints refer to all shifts within a column of the roster to ensure a minimal crew at the ward and to state preferences on larger standard crews [Meyer auf'm Hofe, 1997].

The nurse rostering system uses an iterative search according to Algorithm 1. An initial assignment to all variables is generated in line 1. The loop from line 2 to 7 performs improvement steps. These steps start with a heuristic detection of a part $X' \cup X_T$ of the current roster A that is considered to be responsible for some deficiencies of A (line 3). Fig. 1 presents a subproblem by grey shaded cells that can result from a heuristic to split the chain of night shifts of nurse No. 1. This chain is too long. Line 5 prepares an exhaustive search for better assignments to the variables in $X' \cup X_T$ of the chosen subproblem. A call to a branch-and-bound procedure completes the improvement step in line 6.

As mentioned above, Fig. 1 presents an example for a subproblem $X' \cup X_T$ that may be chosen in order to repair

	Th	Fr	Sa	Su	Mo	Tu	We	Th	Fr	Sa	Su	Mo	Tu	We
	1	2	3	4	5	6	7	8	9	10	11	12	13	14
nurse #1	NS	-	-	NS	NS	NS	NS	NS	NS	-	-	MS	MS	LS
nurse #2	MS	MS	MS	-	DT	DT	-	-	DT	NS	-	-	MS	MS
nurse #3	MS	MS	-	-	MS	MS	MS	MS	MS	-	NS	NS	NS	NS
nurse #4	-	NS	NS	-	-	MS	MS	MS	LS	LS	-	DT	-	DT
nurse #5	DT	DT	-	MS	MS	LS	LS	LS	-	-	MS	MS	LS	MS
nurse #6	LS	LS	LS	-	LS	-	DT	DT	MS	MS	-	-	DT	DT
nurse #7	LS	-	-	LS	LS	LS	LS	LS	-	-	LS	LS	LS	LS

Figure 1. *A roster and a subproblem representing a repair step.*

Algorithm 1: ITERATIVEIMPROVEMENT ($\mathcal{P} = \langle X, D, C, \mathrm{var}, \varphi, \succ \rangle$)

1: Compute an initial assignment A to all variables in X.
2: **loop**
3: Set $X' \cup X_T \subseteq X$ to hold a region where A possibly is suboptimal and X_T forms a tree-structured subproblem.
4: **if** $X' = \emptyset$ **then break**, **end if**
5: Generate an entry $conflicts[x \leftarrow d]$ for each $x \in X' \cup X_T$ and $d \in D$ that holds all conflicts with $A{\downarrow}\, X \setminus (X' \cup X_T)$.
6: If BRANCHANDBOUND($\mathcal{P}$, β $\mathcal{V}_P(A{\downarrow}\,(X \setminus (X' \cup X_T)),\ A{\downarrow}\,(X \setminus (X' \cup X_T),\ X',\ X_T,\ conflicts)$ leads to an improved solution then assign this improvement to A.
7: **end loop**
8: **return** A

a highly relevant deficiency of a schedule. As mentioned above, all constraints are either oriented along the rows or the columns in the roster. As it will be shown later, the constraint graph of this subproblem is a hyper tree. This observation motivates a further investigation of efficient procedures for solving tree-structured constraint problems [Freuder, 1982, Freuder, 1990, Dechter *et al.*, 1990, Freuder and Wallace, 1992]. As a consequence, this paper addresses opportunities to extend branch-and-bound search by the cycle-cutset method [Dechter and Pearl, 1987, Dechter, 1990]. This improved branch-and-bound is then used to perform the the repair step in Algorithm 1. For this reason, line 3 of Algorithm 1 returns two sets of variables to indicate a subproblem: X' and X_T. The variables in X_T are supposed to form a *tree-structured subproblem* that can be solved by efficient algorithms. In contrast, X' is thought to be a cutset: A set of variables that need to have a certain value assigned in order to enable the use of efficient algorithms for tree-structured problems.

This paper suggests to apply efficient algorithms for special constraint problems — for instance of a tree-like structure — as a repair step in iterative search to solve real world problems like nurse rostering. Therefore, the next section briefly illustrates partial constraint satisfaction and tree-structured constraint problems accompanied with the relevance of these definitions for nurse rostering. Starting with reformulations of standard algorithms, the next two sections develop algorithms for solving k-tree structured subproblems in real world applications. Concluding remarks sum up the results.

2 Nurse Rostering as Partial Constraint Satisfaction

Partial Constraint Satisfaction with Fuzzy Constraints

As it is well known, constraint problems consist of constraint variables and their domains which is a set of labels that can be assigned to the variable. Constraints post restrictions on consistent assignments of labels to two or more variables. The standard constraint problem is to assign labels to all variables in such a way that all constraints are satisfied. Things get a bit more complex on partial constraint satisfaction where solutions to a constraint problem are only required to satisfy the constraints as good as possible. Such forms of constraint problems are appropriate to represent all kinds of optimization problems. This section describes a notion of partial constraint satisfaction with fuzzy constraints [Meyer auf 'm Hofe, 2000] that uses a partial ordering of fuzzy sets of constraints to distinguish more from less important conflicts. The standard PCSP [Freuder and Wallace, 1992] considers soft but crisp constraints, i.e. solutions are not necessarily required to satisfy all constraints but all constraints are either completely satisfied or completely violated. In contrast, fuzzy constraints may be *partially* satisfied by a solution.

Let X be the set of variables, D the domain of the variables, and C be the set of constraints that describe constraint problem $\mathcal{P}$. Each constraint $c \in C$ has a set of local variables var_c.

An assignment to the variables in set X' is a set of labelings $\{x \leftarrow d \mid x \in X', d \in D\}$. $A{\downarrow}\, X''$ selects from A the assignments to the variables in X''. Additionally, $A{\downarrow}\, x$ denotes the value that A assigns to variable x.

A function $\varphi_c : D^{\mathrm{var}_c} \to [0;1]$ maps a degree of constraint violation to each assignment A to the local variables var_c. A 0 degree of constraint violation means that A satisfies the constraint perfectly. A 1 degree of constraint violation indicates a complete violation. Degrees between these extremes represent a partial violation.

Since assignments may violate constraints to a degree from 0 to 1, the conflicts of each assignment A can be represented by a fuzzy set $\bar{C}(A)$ where $\varphi_c(A{\downarrow}\ \mathrm{var}_c)$ is the membership of constraint c in this set. This view makes it easy to sum up conflicts by use of fuzzy set union $C_1 = C_2 \sqcup C_3$ where the membership of constraint c to C_1 is the maximum of c's membership to C_2 and C_3.

Finally, a partial ordering $\succ$ among fuzzy sets of constraints describes which conflicts are more important than others. A fuzzy set of constraints C_1 is more important than another fuzzy set of constraints C_2 iff $C_1 \succ C_2$. An assignment A is a solution of constraint problem $\mathcal{P}$ iff an assignment A' with less important conflicts — i.e. $\bar{C}(A) \succ \bar{C}(A')$ — does not exist.

PCSPs according to this definition provide a very flexible formalism to represent problems like nurse rostering. Nurse rostering concerns classical binary constraints — for instance to represent compatibility of consecutive shifts — as well as global constraints which affect a large number of local variables. As an example for such constraints in nurse rostering, consider the APPROX constraint that is for instance used to prefer attendance of a standard crew at the ward and a balanced working time account. The degree of violating this constraint is defined with respect to two parameters f and g, where $f(d)$ maps a weight to each value in the domain D and g is a goal sum of this weights. The degree $\varphi_{f,g}(A)$ of violation a constraint c of this type grows with the distance between the goal sum g and the sum of weights of the values that A assigns to the local variables.

$$\varphi_{f,g}(A) = \frac{|g - \sum_{\{x \leftarrow d\} \in A} f(d)|}{|\mathrm{var}_c| \cdot \max\{f(d) \mid d \in D\}}$$

Let for instance f map a 1 to the morning shift and a 0 to all other shifts. Then an APPROX constraint is appropriate to state a preference on a standard crew of 2 nurses during the morning shift on day 2 in Fig. 1. The constraint has a goal sum of 2 and all constraint variables concerning shifts on day 2 as local variables. Consequence: The degree of constraint violation grows with the difference between the scheduled crew size and the preferred crew size during the time period of the morning shift.

Tree-structured Subproblems

Fig. 2 presents the constraint graph of a constraint problem. The nodes in this graph represent variables. The hyper-arcs represent constraints between the connected variables. The tree-structured subproblem consists of the numbered nodes and constraints. The striped constraint disturbs the tree-structure of the problem and the striped nodes represent a

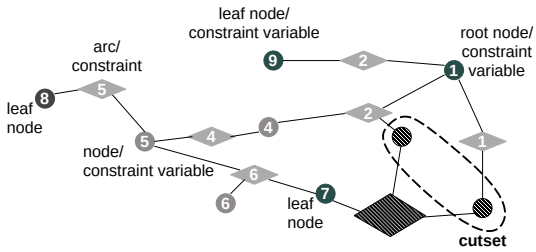


Figure 2. *Components of a tree-structured constraint network.*

so-called cutset: If these variables are labeled with unique values then the striped constraint degenerates to a unary constraint. As a consequence, the rest of the constraint problem has a tree-structure [Dechter and Pearl, 1987, Dechter, 1990].

A constraint graph is *plain* iff two nodes are connected by at most one hyper-arc.

In Fig. 2, the numbers of nodes represent a total ordering $>$ of the nodes. The *width of a node x_1 of the constraint graph with respect to a particular ordering* $>$ is defined to be the number of nodes x_2 with $x_1 > x_2$ where x_1 and x_2 are connected by different arcs (or constraints). The *width of the constraint graph with respect to a particular ordering* $>$ is the largest width of a node. $>$ is called to be the *best ordering in a particular constraint graph* iff the width of the constraint graph with respect to $>$ is minimal. The *width* of a constraint graph (in general) is the width with respect to the best ordering [Freuder, 1982].

A constraint problem whose constraint graph is of width 1 is called a tree-structured problem. The smallest node according to $>$ is the root node. All nodes which are not connected to $>$-larger nodes are leaf nodes. An ordering $>$ proving width 1 is also called tree-ordering if for each constraint the local variable which is the smallest due to $>$ is connected with a smaller variable.

Fig. 2 presents such a tree-structured problem. The numbers of the nodes indicate a tree-ordering. Each node is at most connected with one node of a smaller index.

Consequently, a tree-ordering $>$ can be used also in the nurse rostering application to prove that a subproblem has a tree structure. Since each cell in a roster corresponds with a constraint variable in the constraint problem, Fig. 3 presents a tree-ordering for the subproblem of Fig. 1 numbering the nodes in the constraint graph from 1 (root node) to 21. The constraints on a roster either connect cells in the same row of a roster — maximal and minimal working time, preferred working time, constraints on minimal resting time, and so on — or the same column of a roster — for each required shift type a constraint on minimal and preferred crew size. Hence, all variables referring to cells of the same row or column are connected by constraints. The arrows in Fig. 3 point from each constraint variable x_1 to a variable x_2 iff both are local to the same constraint and $x_1 > x_2$. Since at most one arrow leaves from the same variable, the subproblem has width 1. This small example shows that more

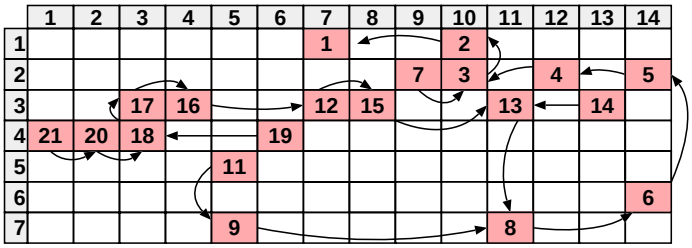


Figure 3. *Constraints connect only variables within the same row or the same column. Consequence: The subproblem's constraint graph has the form of a hyper-tree.*

efficient algorithms for solving tree-structured subproblems are also relevant to improve nurse rostering if they are not restricted to subproblems of a plain constraint graph — in the nurse rostering application many constraints overlap in more than one variable.

3 Cycle-Cutset and Branch-and-Bound

This section introduces an extension of the branch-and-bound with cycle-cutset that is derived directly from the literature. Algorithm 2 presents a version of the branch-and-bound as it is referred by Algorithm 1. The branch-and-bound receives as arguments: The constraint problem $\mathcal{P}$, the bound β, the best yet found solution S, the currently explored assignment A, and the distance δ, that is the fuzzy set of constraints reflecting A's constraint violations. Furthermore, X' and X_T specify which part of $\mathcal{P}$ has to be searched. The variables in X_T are known to form a tree-structured subproblem. The branch-and-bound has to branch over the possible assignments to the variables in X' whereas the variables in X_T will be labeled by a special algorithm[1].

Line 1 leads to a backtracking if the conflicts δ of the currently explored assignment A exceed bound β. Then, line 2 determines whether the branch-and-bound has to perform a branching or not. If both X' and X_T are empty, then A is a new solution. Hence, line 4 returns this new solution and its conflicts. If X_T represents a non-trivial tree-structured subproblem, line 6 calls Algorithm 4 SOLVETREE to solve the remaining problem. If X' is not empty, line 3 chooses a variable and the loop in line 10 loops over all admissible assignments to this variable. Then, line 12 calls constraint propagation to find out which conflicts arise on the new assignment $x \leftarrow d$. Line 14 implements the final step in branching: A recursive call of the branch-and-bound including the new assignment and a new distance.

This version of the branch-and-bound uses constraint propagation to detect conflicts which conclude from the currently explored assignments. Array $conflicts$ stores for each assignment to a yet unlabeled variable the corresponding detected conflicts. Algorithm 3 presents a generic procedure for the propagation of fuzzy constraints: The min-max-propagation [Snow and Freuder, 1990, Dubois *et al.*, 1993]. This procedure loops over all admissible assignments to the local variables of the constraint (line 4). For each of these "tuples", fuzzy set $tupleConflicts$ holds the propagated constraint with a membership according to the tuple's degree of violating the constraint. Additionally, $tupleConflicts$ is loaded with the conflicts that have been detected in advance to each of the assigned values (line 7). After this procedure, $tupleConflicts$ holds for each constraint the maximal degree of constraint violation that follows either from the propagated constraint or that has been detected in advance for one of the assigned values. Then, line 10 collects for each labeling of a single variable the conflicts according to the best tuple A' it appears in. Finally, $conflicts$ is loaded with the newly detected conflicts. Provided that φ_c can be computed efficiently, the effort for the min-max-propagation grows with $|D|^{|\mathrm{var_c}|}$ which is the number of tuples A'.

Algorithm 4 TREESOLVE finds optimal solutions to tree-structured problems with $|C|$ calls of algorithms for constraint propagation. For now assume, that PROPAGATE is synonymous with the max-min-propagation according to Algorithm 3. The basic idea is to propagate the conflicts from the leave nodes to the root node. Consider again Fig. 2 as an example. The loop over line 2 considers at first leaf node 7. Algorithm 5 DIRECTEDPROP is called to propagate the constraints linking node 7 as a root node of a subtree to the branches of the subtree. However, node 7 is a leaf node and, consequently, DIRECTEDPROP has nothing to do. This situation changes on node 5 representing variable x_5. After DIRECTEDPROP on this variable, the $conflicts$ of the labels of variable x_5 are set according to the best opportunity to label the variables x_5 to x_8 forming the subtree below x_5. It can be shown that $conflicts[x_5 \leftarrow d]$ comprises under these circumstances exactly the conflicts of the optimal assignment to the variables in the subtree that also assigns d to x_5. The condition for constraint propagation in DIRECTEDPROP guarantees that constraints of deeper subtrees will be considered before constraints of higher subtrees.

After leaving the propagation phase, the loop starting at line 5 collects labelings to build an optimal solution in the same manner as the branch-and-bound algorithm: Select a new labeling for a variable according to the yet known $conflicts$ and propagate the consequences of this new labeling by the same constraint propagation as it is used in the branch-and-bound to find new conflicts.

Hence, tree-structured problems can be solved efficiently,

[1] The literature on cycle-cutset assumes an algorithm where X_T needs to be computed for any branch in the search tree.

Algorithm 2: BRANCHANDBOUND($\mathcal{P}, \beta, S, \delta, A, X', X_T, conflicts$)

$\mathcal{P}$ is the constraint problem. β is the bound: The fuzzy set of constraints representing the conflicts of the best yet found solution S. Initially, the bound maps full membership to all constraints and S is the empty set. A is the partial assignment describing the current branch of the search tree that is initially the empty set. δ is the fuzzy set of constraints that represents known conflicts concluding from assignment A. X' is the set of variables to be labeled. $X_T \cap X' = \emptyset$ is a tree-structured subproblem. $conflicts$ describes the known conflicts. The result of the algorithm is a tuple $\langle S, \beta \rangle$ representing either the old solution or a new improved one.

1: **if** not $\beta \succ \delta$ **then return** $\langle S, \beta \rangle$, **end if**.
2: **if** $X' = \emptyset$ **then**
3: **if** $X_T = \emptyset$ **then**
4: **return** $\langle A, \delta \rangle$.
5: **else**
6: **return** SOLVETREE($\mathcal{P}, \beta, \delta, A, X_T, conflicts$) .
7: **end if**
8: **else**
9: Choose an $x \in X'$.
10: **for all** $d \in D$ with $\beta \succ conflicts[x \leftarrow d]$ **do**
11: **for all** $c \in C$ with $x \in \mathrm{var}_c$ **do**
12: $conflicts \leftarrow$PROPAGATION($c, \beta\, \delta \sqcup conflicts[x \leftarrow d], A \cup \{x \leftarrow d\}, conflicts$), where Propagation is one of the procedures for constraint propagation described below.
13: **end for**
$\langle S, \beta \rangle \leftarrow$ BRANCHANDBOUND($\mathcal{P}, \beta, \delta \sqcup conflicts[x \leftarrow d]$,
14: $A \cup \{x \leftarrow d\}, X' \setminus \{x\}, X_T,$
$conflicts$).
15: **end for**.
16: **return** $\langle S, \beta \rangle$.
17: **end if**

Algorithm 3: MINMAXPROPAGATION($c, \beta, \delta, A, conflicts$)

c is the constraint to be propagated. Argument A' is a partial assignment that may be used for instance to describe the branch of a search tree. $conflicts[x \leftarrow d]$ is a fuzzy set of constraints that holds the conflicts that follow from assigning label d to variable x.

1: **for all** $x \in \mathrm{var}_c$ and $d \in D$ **do**
2: Add all constraints with membership 1 to $bestConflicts[x \leftarrow d]$.
3: **end for**.
4: **for all** $A' \in D^{\mathrm{var}_c}$ composed of admissible labelings **do**
5: Set $tupleConflicts$ to be the fuzzy set mapping membership $\varphi_c(A')$ to constraint c.
6: **for all** $x \in \mathrm{var}_c$ **do**
7: $tupleConflicts \leftarrow tupleConflicts \sqcup conflicts[x \leftarrow A'{\downarrow}\, x]$.
8: **end for**.
9: **for all** $x \in \mathrm{var}_c$ **do**
10: **if** $bestConflicts[A'{\downarrow}\, \{x\}] \succ tupleConflicts$ **then**
11: $bestConflicts[A'{\downarrow}\, \{x\}] \leftarrow tupleConflicts$.
12: **end if**.
13: **end for**.
14: **end for**.
15: **for all** $x \in \mathrm{var}_c$ and $d \in D$ **do** $conflicts[x \leftarrow d] \leftarrow bestConflicts[x \leftarrow d]$ **end for**.

A labeling $x \leftarrow d$ is admissible iff either $(x \leftarrow d) \in A$ or A does not assign a value to variable x and $\beta \succ \delta \sqcup conflicts[x \leftarrow d]$.

if the constraint graph is plain and propagation of the constraints is efficient. Academic papers often concentrate on binary constraints since these constraints guarantee an efficient max-min-propagation.

Fig. 4 shows the effect of introducing cycle-cutset into the branch-and-bound (bb+cycle-cutset) and of using this algorithm for steps of repair (iterative cycle-cutset) compared to the branch-and-bound with constraint propagation but without using TREESOLVE (bb). The diagrams show an average performance on randomly generated PCSPs of a plain constraint graph and with binary and crisp constraints of low satisfiability. The constraints have a randomly chosen weight and the preference $\succ$ is defined according to the weight sum of violated constraints [Freuder and Wallace, 1992]. The curves show the decrease in the weight of constraints violated by the best yet found solution over

Algorithm 4: TREESOLVE($\mathcal{P}, \beta, \delta, A, X_T, conflicts$)

Let $>$ be a tree-ordering on the variables of the subproblem. Let $C_T \leftarrow \{c \mid |\text{var}_c \cap X_T| \geq 2\}$ be the constraints connecting variables of the tree-structured subproblem, i.e. the set of constraints oft he tree-structured subproblem.

1: **for all** $x \in X_T$ ordered by $>$ starting with the largest variable **do**
2: DIRECTEDPROP($x, C_T, \beta, \delta, A, conflicts$).
3: **end for**.
4: **if** not $\beta \succ$ **then return** $\langle \emptyset, \beta \rangle$, **end if**
5: **for all** $x \in X$ ordered by $>$ starting with the smallest variable (root node) **do**
6: Choose a value $d \in D$ with minimal $\delta \sqcup conflicts[x \leftarrow d]$.
7: **for all** $c \in C$ with $x \in \text{var}_c$ **do** PROPAGATION($c, \beta, \delta \sqcup conflicts[x \leftarrow d], A, conflicts$) **end for**.
8: $\delta \leftarrow \delta \sqcup conflicts[x \leftarrow d]$.
9: **end for**.
10: **return** $\langle A, \delta \rangle$.

Algorithm 5: DIRECTEDPROP($x', C_T, \beta, \delta, A, conflicts$)

1: **for all** $c \in C_T$ where x' is minimal local variable of c due to $>$ **do**
2: MINMAXPROPAGATION($c, \beta, \delta, A, conflicts$).
3: **end for**.
4: **return** $conflicts$.

time. Diagram a) shows the performance on nearly tree-structured problems: All plain but connected graphs with 10 nodes and 9 arcs form a tree. As a consequence, both algorithms using cycle cutset converge very quickly on the optimal solution. With increasing number of the constraints, the branch-and-bound with cycle-cutset becomes worse than branch-and-bound without cycle-cutset since the solved problems loose their tree-like structure. In contrast, an iterative search exploiting cycle-cutset converges still far more quickly on optimal solutions than the branch-and-bound. Apparently, especially the integration into iterative search makes cycle-cutset a really attractive method for solving constraint problems.

However, algorithm TREESOLVE is only applicable to subproblems of a *plain* constraint graph. Real world problems imply only in very rare cases plain constraint graphs since these applications depend very often on constraints of large arity (global constraints) that often overlap in more than one variable. The APPROX constraint of section 2 provides an example for such large arity constraints.

4 Solving Tree-structured Problems with Shallow Propagation

Algorithm MINMAXPROPAGATION looks for optimal support with respect to the propagated constraint *and* previously recognized conflicts. Both requirements in combination make it hardly possible to implement this kind of propagation efficiently for special types of large arity constraints — but efficient propagation of large arity constraints is one of the key factors for the success of constraint-based problem solving. Fortunately, Algorithm 2 BRANCHANDBOUND can make use of a reduced form of constraint propagation that distinguishes only admissible from non-admissible labelings. A labeling $x \leftarrow d$ is called to be admissible iff it is an element of assignment A describing the currently explored branch of the search tree or x is a yet unlabeled variable and the currently known conflicts of $x \leftarrow d$ do not exceed the bound β [Meyer auf'm Hofe, 1999, Meyer auf'm Hofe, 2000]. This *shallow propagation* determines conflicts with a constraint c according to possible admissible assignments to the local variables of c. On the one hand, such *shallow propagation* is useful to detect conflicts early in a kind of extended forward checking as performed in line 12 of Algorithm 2. On the other hand, efficient algorithms can be found for many relevant constraints. As an example consider Algorithm 6 SHALLOWPROPAPPROX presenting a shallow propagation for the constraints of type APPROX that has been defined in section 2.

Let A be the partial assignment describing the currently valid branch in the search tree. A's degree of violating a constraint of type APPROX is determined according to the distance between $\sum_{x \in \text{var}_c} f(A{\downarrow}\, x)$ and a goal sum g. Algorithm 6 determines the conflicts for currently unlabeled variables by a simple procedure that at first sums up the minimal sum $f_\Sigma^{\min}$and the maximal sum $f_\Sigma^{\max}$ of weights resulting from f on admissible values (in the loop starting at line 2). The second loop starting at line 6 uses these results to determine for each admissible labeling $x \leftarrow d$ the minimal and maximal sum of weights $f_{x \leftarrow d}^{\min}$ and $f_{x \leftarrow d}^{\max}$ that can result from extending $x \leftarrow d$ with admissible labelings. A complete satisfaction of the constraint is considered to be possible if the goal sum g lies between these values. Otherwise, an optimistic estimate on the degree of constraint violation that results from assigning d to x is computed from $f_{x \leftarrow d}^{\min}$ or $f_{x \leftarrow d}^{\max}$. The effort for this algorithm grows linear with the size of the domain D and the arity $|\text{var}_c|$ and is, thus, very efficient.

Such algorithms detect conflicts with partial assignments as required in line 12 of Algorithm 2 and in line 7 of Algorithm 4. However, shallow propagation is not appropriate for the task of collecting all conflicts in a subproblem as required in line 2 of Algorithm 5 since shallow prop-

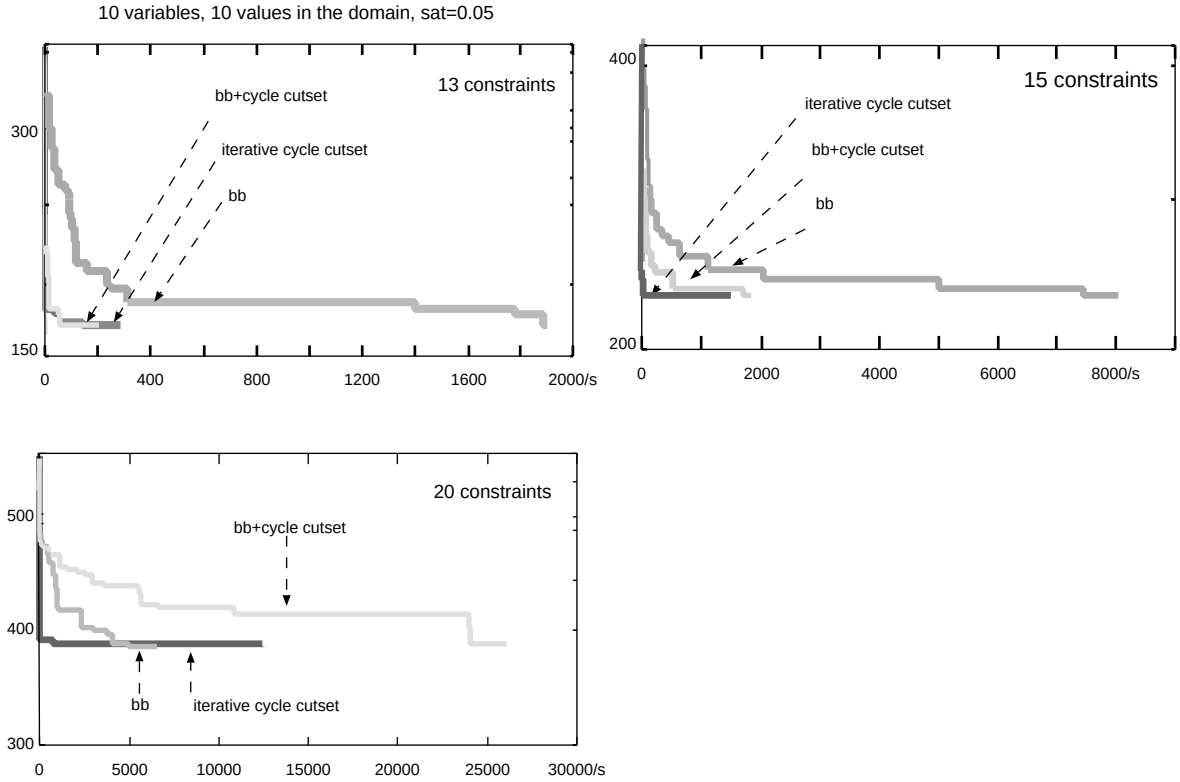


Figure 4. The basic effect of cycle-cutset as step of repair within iterative improvement.

agation only distinguishes between admissible and non-admissible labelings. These algorithms fail, thus, to propagate conflicts detected in deeper subtrees to higher nodes of a tree-structured problem. As a consequence, the call to DIRECTEDPROP in Algorithm 4 shall be replaced by Algorithm 7 DIRECTEDCLUSTERPROP, that is able to solve tree-structured subproblems which form a hyper tree where hyper-arcs are allowed to share more than one variable.

Algorithm 7 DIRECTEDCLUSTERPROP is based on Freuder's idea to solve k-structured trees [Freuder, 1990]. The idea is to group the variables of a constraint problem into clusters in such a way that the arcs between the clusters form a tree. These clusters form subproblems that can be treated as single constraint variables where the set of all solutions to a cluster can be considered as the domain. Now, algorithms on solving tree-structured problems are able to combine solutions to the clusters in a consistent way. The complexity of this algorithm depends on the size k of the largest cluster.

Algorithm 7 applies this idea in a two phase procedure. The first phase spanning over the lines 1 to 8 determines the cluster containing variable x' where x' is the smallest variable due to $>$. Line 1 collects constraints with x' as smallest local variable. As in Algorithm 5 DIRECTEDPROP, these constraints have to be propagated in order to collect the conflicts within the subtree below x'. The following loop starting with line 3 adds all constraints (and their local variables) to the cluster that overlap with more than one variable. The procedure exits immediately with result $conflicts$ if x' is either not the smallest local variable of a constraint or x' is not the smallest variable of the cluster. In these cases, DIRECTEDCLUSTERPROP is a *no operation* because the cluster containing x' has to be propagated later on in order to follow the tree-structure of the connections between the clusters. At the end of the first phase, X' and C' are the components of a cluster where x' is the smallest variable — and, therefore, the interface to the clusters which are higher in the tree. This cluster contains all constraints with x' as local variable and additionally all constraints that overlap in more than one variable.

The goal of phase two is to collect for each assignment $x' \leftarrow d$ to variable x' the conflicts of the best assignment to the variables X' of the clusters that labels x' with d since x' is the only variable that may also be part of clusters which are higher in the tree. This phase starts with line 10, an enumeration of all admissible assignments to the variables in $X' \setminus \{x'\}$. This loop collects all conflicts that have been detected for the current assignment into $tupleConflicts$. Shallow propagation of all constraints in C' adds all conflicts with constraints of the current cluster. $bestConflicts$ is set according to the result of the best assignment to the variables in the cluster. As a final result, $conflicts$ is set to the least important conflicts that have been detected. The effort for this algorithm grows with $|C'| \cdot |D|^{|X'|-1}$ multiplied with the effort for constraint

Algorithm 6: SHALLOWPROPAPPROX($c, \beta, \delta, A,$ *conflicts*)

As mentioned in the text, this constraint has a goal sum g and a function f as a special attribute that maps each value in D to a real number. Let A be an assignment to the local variables. Then, A's degree of violating the constraint is defined as follows:

$$\varphi_{f,g}(A) = \frac{|g - \sum_{\{x \leftarrow d\} \in A} f(d)|}{|\mathrm{var}_c| \cdot \max\{f(d) \mid d \in D\}}$$

1: $f_\Sigma^{\max} \leftarrow 0$. $f_\Sigma^{\min} \leftarrow 0$.
2: **for all** $x \in \mathrm{var}_c$ **do**
3: $f_\Sigma^{\max} \leftarrow f_\Sigma^{\max} + \max\{f(d) \mid x \leftarrow d \text{ is admissible}\}$.
4: $f_\Sigma^{\min} \leftarrow f_\Sigma^{\min} + \min\{f(d) \mid x \leftarrow d \text{ is admissible}\}$.
5: **end for**.
6: **for all** $x \in \mathrm{var}_c$ and $x \leftarrow d$ is admissibe **do**
7: $f_{x \leftarrow d}^{\max} \leftarrow f_\Sigma^{\max} - \max\{f(d') \mid x \leftarrow d' \text{ is admissible}\} + f(d)$.
8: $f_{x \leftarrow d}^{\min} \leftarrow f_\Sigma^{\min} - \min\{f(d') \mid x \leftarrow d' \text{ is admissible}\} + f(d)$.
9: **if** $f_{x \leftarrow d}^{\max} < g$ **then**
10: Add c with a degree of $\frac{g - f_{x \leftarrow d}^{\max}}{|\mathrm{var}_c| \cdot \max\{f(d') \mid d' \in D\}}$ to $conflicts[x \leftarrow d]$.
11: **else if** $f_{x \leftarrow d}^{\min} > g$ **then**
12: Add c with a degree of $\frac{f_{x \leftarrow d}^{\min} - g}{|\mathrm{var}_c| \cdot \max\{f(d') \mid d' \in D\}}$ to $conflicts[x \leftarrow d]$.
13: **end if**.
14: **end for**.

Algorithm 7: DIRECTEDCLUSTERPROP($x', C_T, \beta, \delta, A,$ *conflicts*)

1: $C' \leftarrow \{c \in C_T \mid x \text{ is minimal local variable of } c \text{ due to } >\}$. $X' \leftarrow \bigcup_{c \in C'} \mathrm{var}_c$.
2: **if** $C' = \emptyset$ **then return** $conflicts$, **end if**
3: **for all** $c \in C_T$ **do**
4: **if** $|X' \cap \mathrm{var}_c| \geq 2$ **then**
5: **if** x' is not minimal local variable due to $>$ **then return** $conflicts$, **end if**
6: Add all variables in var_c to X'. Insert c into C'.
7: **end if**
8: **end for**
9: **for all** $d \in D$ **do** Add all constraints with membership 1 to $bestConflicts[x' \leftarrow d]$. **end for**.
10: **for all** $A' \in D^{X' \setminus \{x'\}}$ of admissible labelings **do**
11: Initialize $tupleConflicts$ to hold an empty set for each $x \leftarrow d$ with $x \in \mathrm{var}_c \setminus X'$ and d $\in D$.
12: **for all** x $\in X'$ and $d \in D$ **do** $tupleConflicts[x' \leftarrow d] \leftarrow tupleConflicts[x' \leftarrow d] \sqcup conflicts[A'\downarrow \{x\}]$. **end for**.
13: **for all** $c \in C'$ **do** $tupleConflicts \leftarrow$ SHALLOWPROP($c, \beta, \delta, A \cup A',$ $tupleConflicts$). **end for**
14: **for all** $d \in D$ **do**
15: **if** $bestConflicts[x' \leftarrow d] \succ tupleConflicts[x' \leftarrow d]$ **then** $bestConflicts[x' \leftarrow d] \leftarrow tupleConflicts[x' \leftarrow d]$. **end if**.
16: **end for**.
17: **end for**.
18: **for all** $d \in D$ **do** $conflicts[x' \leftarrow d] \leftarrow bestConflicts[x' \leftarrow d]$ **end for**.

propagation when only one local variable is not labeled.

Refer again to Fig. 3: The size of the subproblem is 21. The largest cluster in this nurse rostering example is of size 6 (row for nurse 3). This means that rather large subproblems can be optimized within iterative repair at comparably low costs. The effort for the large repair step of Fig. 3 should be pretty much the same as the effort for a sequence of repair steps each treating a single cluster by the branch-and-bound — but the results will differ in most cases. The application of DIRECTEDCLUSTERPROP conducts a global optimization whereas each step in the sequel only optimizes each cluster with respect to the shift assignments that are currently valid in the other clusters.

5 Conclusion

This paper suggests to apply efficient algorithms for special constraint problems — and exhibit for instance a k-tree structure — to real world applications. An example from a nurse rostering domain as well as first empirical results on randomly generated constraint problems illustrated the potential of such algorithms to improve the performance of repair steps in search by iterative improvement.

The paper sketched necessary modifications of algorithms taken from the literature in order to integrate special purpose algorithms into state of the art optimization by branch-and-bound extended with constraint propagation.

The suggested use of efficient algorithms within iterative

improvement raises new questions on opportunities to control the search. Iterative improvement as presented in this paper is based on the heuristic detection of subproblems in the application that are responsible for suboptimal solutions. The suggested extensions reduce the effort for many extensive repair steps. As a consequence, further strategies for search control have to consider both: Assumptions on reasons for deficiencies in the current solution *and* assumptions on the effort for finding an improvement whereas the latter depends strongly on the applicability of efficient algorithms.

REFERENCES

[Dechter and Pearl, 1987] Rina Dechter and Judea Pearl. The cycle-cutset method for improving search performance in AI applications. In *Proceedings of the 3rd IEEE Conference on AI Applications*, Orlando, FL, 1987.

[Dechter *et al.*, 1990] Rina Dechter, Avi Dechter, and Judea Pearl. Optimization in constraint networks. In R. M. Oliver and J. Q. Smith, editors, *Influence Diagrams, Belief Nets and Decision Analysis*, pages 411–425. John Wiley & Sons, 1990.

[Dechter, 1990] Rina Dechter. Enhancement schemes for constraint processing: Backjumping, learning, and cutset decomposition. *Artificial Intelligence*, 41(3):273–312, 1990.

[Dubois *et al.*, 1993] Didier Dubois, Hélène Fargier, and Henri Prade. The calculus of fuzzy restrictions as a basis for flexible constraint satisfaction. In *Proc. of the 2nd IEEE International Conference on Fuzzy Systems*, pages 1131–1136, San Francisco, CA, 1993.

[Freuder and Wallace, 1992] Eugene C. Freuder and Richard J. Wallace. Partial constraint satisfaction. *Artificial Intelligence*, 58:21–70, 1992.

[Freuder, 1982] Eugene C. Freuder. A sufficient condition for backtrack-free search. *Journal of the ACM*, 29(1):24–32, January 1982.

[Freuder, 1990] Eugene C. Freuder. Complexity of k-tree structured constraint satisfaction problems. In *Proceedings of the AAAI*, pages 4–9, 1990.

[Meyer auf'm Hofe, 1997] Harald Meyer auf'm Hofe. ConPlan/SIEDAplan: Personnel assignment as a problem of hierarchical constraint satisfaction. In *PACT-97: Proceedings of the Third International Conference on the Practical Application of Constraint Technology*, pages 257–272, London, UK, April 1997. Practical Application Expo.

[Meyer auf'm Hofe, 1999] Harald Meyer auf'm Hofe. *Kombinatorische Optimierung mit Constraintverfahren — Problemlösung ohne anwendungsspezifische Suchstrategien*. Dissertation, Fachbereich Informatik der Universität Kaiserslautern, vorgelegt November 1999.

[Meyer auf'm Hofe, 2000] Harald Meyer auf'm Hofe. Nurse rostering as constraint satisfaction with fuzzy constraints and inferred control strategies. In Eugene C. Freuder and Rick J. Wallace, editors, *Volume on Constraint Programming and Large Scale Optimisation Problems*, DIMACS Volume Series, 2000. In preparation.

[Snow and Freuder, 1990] Paul Snow and Eugene C. Freuder. Improved relaxation and search methods for approximate constraint satisfaction with a maximin criterion. In *Proc. of the 8^{th} biennal conf. of the canadian society for comput. studies of intelligence*, pages 227–230, May 1990.

Knowledge-Based Control of Decision-Theoretic Planning – Adaptive Planning Model Selection –

Jun Miura and **Yoshiaki Shirai**
Dept. of Computer-Controlled Mechanical Systems, Osaka University
Suita, Osaka 565-0871, Japan
email: jun@mech.eng.osaka-u.ac.jp
URL: http://www-cv.mech.eng.osaka-u.ac.jp/~jun

Abstract.

This paper proposes a new planning architecture for agents operating in uncertain and dynamic environments. Decision-theoretic planning has been recognized as a useful tool for reasoning under uncertainty; it calculates an optimal plan using a given *planning model* (state set, action set, probability distributions over possible state transitions, and utility function). In a dynamic environment, however, the current situation may be different from what an agent expects and the current planning model may not be feasible. It is, therefore, important for an agent to continuously examine the situation and to use an appropriate planning model. For this purpose, we propose to employ a knowledge-based meta-level reasoner to on-line select an appropriate planning model for an object-level decision-theoretic planning, based on the given knowledge of classification of the situation. This architecture could also be effective in reducing the computational cost of decision-theoretic planning by limiting the search space according to the situation. Two applications of the architecture to a dynamic robot planning and to a decision-making in highway driving show the generality and the usefulness of the architecture.

1 Introduction

To design planning algorithms for an agent that operates in the real world, we need to consider the following two issues: uncertainty and limitation of computational resources. Various activities of an agent such as sensing and motion inherently include uncertainty; an agent's computational power is definitely limited. It is, therefore, important for an agent to cope with uncertainties without largely increasing the computational cost.

Decision-theoretic planning [2] has been recognized as a useful tool for reasoning under uncertainty; it is usually defined by the following:

- state set $\mathcal{S}$.
- action set $\mathcal{A}$.
- probability distribution over the possible state transitions for executing action $a \in \mathcal{A}$ in state $s \in \mathcal{S}$. Exogenous changes are also included here, if any.
- utility function to evaluate each state or each state-action pair.

In this paper, we collectively call them a *planning model*. Under this model, an agent usually determine an action (or action sequence) which maximizes the expected utility.

Many works have adopted decision-theoretic planning to planning tasks under uncertainty (e.g., [11] [6] [13]). These works, however, assume that the environment is static.

Decision-theoretic planning is usually costly because it has to consider multiple outcomes of actions and the planning cost often increases exponentially to the search depth. This is a drawback when used in a dynamic environment where the allocated time for planning is limited. Recently Markov decision processes (MDPs) have been attracting much interests as a basic representation for planning under uncertainty [7]. Although several approaches (e.g., [4]) have been developed to efficiently obtain optimal policies for MDP problems, they still seem inappropriate for large-sized planning problems under time pressure.

One approach to reducing the computational cost is to properly control the allocation of computing resources to each decision-theoretic planning activity. Many works have recently been focusing on the concept of *limited rationality* [15], in which the cost of planning is explicitly considered and computational resources for object-level planning is allocated so that the overall utility including both plan efficiency and planning cost is maximized. Some of examples are: flexible computation [10], decision-theoretic meta-level control of (object-level) reasoning [15], and expectation-driven iterative refinement (EDIR) using anytime algorithms [3] [14].

These works are mainly interested in optimal allocation of computing resources *within a given planning model*. In a dynamic environment, however, we have to cope with the change of situation by switching planning models. Since a planning using a wrong model may lead to a fatal situation, it is important to frequently examine if the current model is fit to the current situation and to switch to an appropriate model, whenever necessary.

It could be possible to have a very large model which covers all possible situations. However this approach may not desirable due to the following two reasons. First, adopting a large model is computationally expensive in both model generation and model utilization because the number of transition relationships between states grows exponentially to that of states. Second, considering all possibility at once could sometimes hide the underlying structure of a planning problem.

Figure 1. Examining the whole problem space or examining only a part of the space.

An example of such a case will be shown in the next section.

Based on the above discussion, we propose to put a knowledge-based model selector on top of an ordinary decision-theoretic object-level planner. Given the knowledge of the structure of the planning problem, the model selector selects an appropriate planning model, thereby directing the efforts only to a limited, effective (or correct) computation (see Figure 1).

Many layered architectures have been proposed for controlling autonomous robots. Gat [9] proposed a three-level control architecture. In his architecture, called ATLANTIS, the controller is responsible for controlling primitive activities, which are usually reactive sensorimotor processes; the deliberator controls time-consuming computational activities such as planning and world model maintenance; the sequencer coordinates such various activities by initiating and terminating them according to the current goal and situation. Pell et al. [8] proposed a similar architecture for autonomous spacecraft. Such works mainly discuss how to integrate deliberative and reactive activities and deal with the level of planning and executing actions. Since this level corresponds to the decision-theoretic layer in our case, our approach of putting a knowledge-based model selector can also be adopted to these control architectures.

The rest of the paper is organized as follows. Section 2 shows a simple example in which the analysis of the structure of the planning problem is important. Section 3 describes the proposed planning architecture. Section 4 describes the application of the proposed architecture to a mobile robot planning problem in a dynamic environment. Section 5 describes the application to a tactical reasoning in driving used for an intelligent driver assistance system. Section 6 summarizes the paper and discusses future works.

2 Importance of Knowledge of Problem Structure: A Simple Example

This section shows a simple example in which considering the problem structure is important. Figure 2 shows a situation where we are going from city A to city B. We select the route among the three, R_1, R_2, and R_3. The degrees of congestion of R_1 and R_3 are dependent on the current situation, while that of R_2 is constant. We suppose there are two *states*: s_1 is the state that R_1 is more congested than R_3; s_2 is the contrary state. We assume the loss table shown in Table 1, where C_1 and C_2 are losses (it could be the necessary time of travel) imposed by taking a specific combination of the state and the route.

We here suppose that which state actually occurs depends

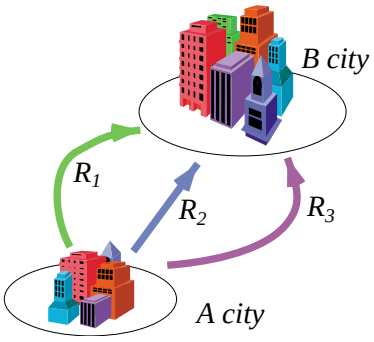


Figure 2. A route selection problem.

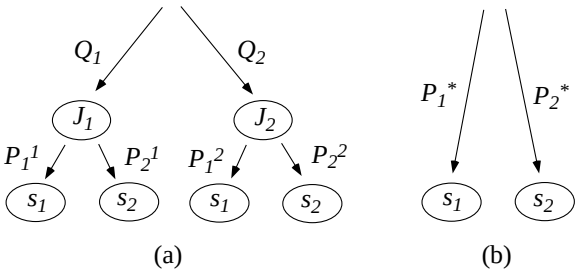


Figure 3. Structure of probabilistic inference.

Table 1. A loss table.

	route to take		
	R_1	R_2	R_3
S_1	C_1	C_2	0
S_2	0	C_2	C_1

on some other factors such as the time of a day. For example, we can consider the case where s_1 is more likely to occur in the morning (we call this situation J_1) and s_2 is in the afternoon (situation J_2). Let P_i^j be the probability that state i occurs in situation J_j and Q_j be the probability of situation J_j (see Figure 3(a)). P_i^j can be regarded as the *model* of situation J_j. If we do not know this hidden structure of the problem (i.e., situation decomposition into J_1 and J_2), we have to use the probabilities of the states directly (see Figure 3(b)), which would probably be estimated from the samples for a whole day. Let P_1^* and P_2^* be such a probability of each state; P_i^* is given by

$$P_i^* = Q_1 P_i^1 + Q_2 P_i^2 \qquad (i = 1, 2)$$

As an example, consider the following parameters: $C_1 = 50$, $C_2 = 20$, $P_1^1 = P_2^2 = 0.8$, $P_2^2 = P_1^2 = 0.2$. Using these values, we summarize in Table 2 the expectation of taking each route in the three cases: (1) the situation is known to be J_1 (i.e., $Q_1 = 1, Q_2 = 0$), (2) the situation is known to be J_2 ($Q_1 = 0, Q_2 = 1$), and (3) the *unknown* situation where J_1

Table 2. Expected loss of taking each route. Underlined is the optimal.

		$E[R_1]$	$E[R_2]$	$E[R_3]$
situation	J_1	40	20	<u>10</u>
	J_2	<u>10</u>	20	40
	unknown	25	<u>20</u>	25

and J_2 are equally likely to occur ($Q_1 = Q_2 = 0.5$). From the table, we can see that if we know the current situation, we can perform planning only for the situation, thereby obtaining a more efficient plan.

If we know the probability of each model, we can select an optimal action, for example, which minimizes the expected loss. Suematsu and Hayashi [16] propose an algorithm to calculate an optimal policy which maximize the expectation of the average reward per step given a set of candidate models and the probabilistic distribution over the set. However such an approach could be computationally expensive, because basically all possibilities (models) have to be examined.

What we would like to stress here are that for efficient (or tractable) model construction and utilization, we should use as much knowledge of the problem structures as possible, and that a planning architecture should be capable of effectively utilizing such knowledge. These are the strong motivation for us to propose the three-level planning architecture.

3 Three-Level Planning Architecture

Based on the above discussion, we propose a three-level control architecture shown in Figure 4. Each level is considered to operate in parallel with the others. The functions of each level is as follows.

Knowledge-based model selector

This level continuously examines the environment and classifies the current situation into one of known categories. Based on the selected category, the corresponding planning model is selected and given to the next level. As a model selector, we can use any classifier; for example, a Bayesian classifier [5] can be used to assess the probability of each category and the reliability of each planning model in the current situation. Concerning the applications presented in this paper, in Section 4, we use a simple frequency-based classifier; in section 5, on the other hand, we use a hand-coded state-transition graph-based classifier.

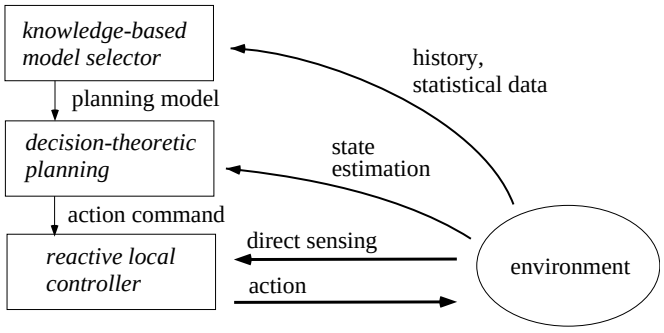


Figure 4. Three-level planning architecture.

Base-level decision-thoretic planner

This level performs planning using the given planning model and the state estimation result to select the best action which minimizes the expected utility, and sends the selected action to the next level. Any decision-theoretic planners can be adopted as long as they response to the dynamics of the environment reasonably quickly. An appropriate example is Real-time Dynamic Programming (RTDP) [1] which on-line generates a decision tree with a limited depth.

Reactive local controller

This level has a set of actions, each of which is realized as a local sensory-action feedback loop. The upper-level decision-theoretic planner selects an action and this level executes it. In addition, this level occasionally handles emergency situations; in that case, this level overrides the upper levels.

4 Example Domain 1: Mobile Robot Planning in Dynamic Environmen t

This section describes an application of the proposed planning architecture to a mobile robot planning problem in a dynamic environment (see Figure 5).

There is a mobile robot and a moving obstacle. The robot and the obstacle have their own destination and the robot does not know the obstacle's destination. The task of the robot is to reach the destination as early as possible without colliding with the obstacle. The robot has several planning models; only the knowledge of the destination of the obstacle is different in these models. The robot uses one of the models to predict the future movement of the obstacle for decision-theoretic planning.

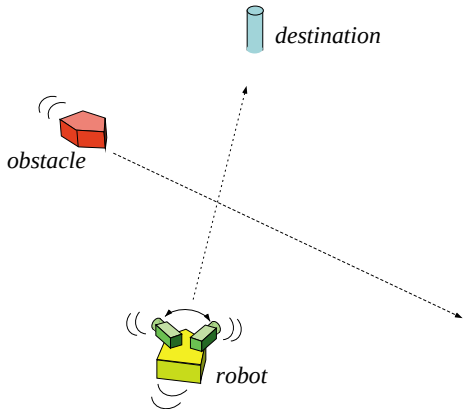


Figure 5. Example problem in dynamic environment.

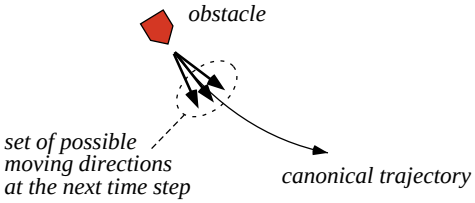


Figure 6. Motion uncertainty model of obstacle.

4.1 Model of Obstacle Motion

Obstacle motion is modeled by its *canonical* trajectory to the destination and motion uncertainty around the trajectory (see Figure 6). We represent the motion uncertainty by a set of possible moving directions at the next time step and the uniform probabilistic distribution over them. We repeatedly apply this uncertainty model to predicting the obstacle position in a near future (a few time steps).

4.2 Decision-Theoretic Robot Motion Selection

Each planning model is composed of the following:

- a state is represented by the position and the velocity of the obstacle and those of the robot.
- an action is the motion (i.e., the moving direction and the speed) of the robot at the next time step.
- a probabilistic distribution is calculated over the possible next position of the obstacle using the motion model (canonical trajectory to a destination).
- a utility function to evaluate the expected time of the robot reaching its destination.

The decision-theoretic planner repeats the cycle of estimating the current state (measuring the position of the obstacle), searching for the best next action, and issuing a command to the controller. The search is performed as follows. First the robot predicts the possible pairs of the position and the velocity of the obstacle and their probabilities at the time a few steps later from the current time by using the motion uncertainty model. For each pair of predicted obstacle position/velocity and robot position, assuming that the obstacle position will diverge around the canonical trajectory (see Figure 7), the robot calculates the time to the destination using our path planner, which generates a minimum-time collision-free path in the time space (see Figure 8). The robot selects the best next action which minimizes the expected time to the destination.

4.3 Selecting Motion Model of Obstacle

The top-level knowledge-based model selector estimates the motion model of the obstacle (i.e., planning model) from the history of obstacle motion. At present, we have tested the following two types of model selectors.

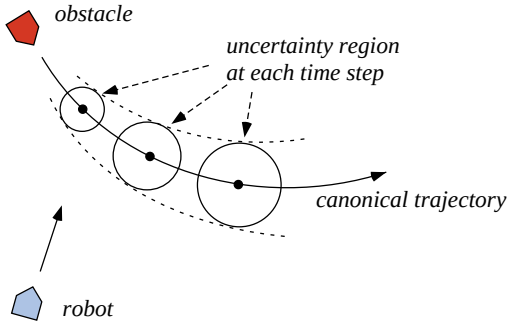


Figure 7. Uncertainty evolution model of obstacle motion.

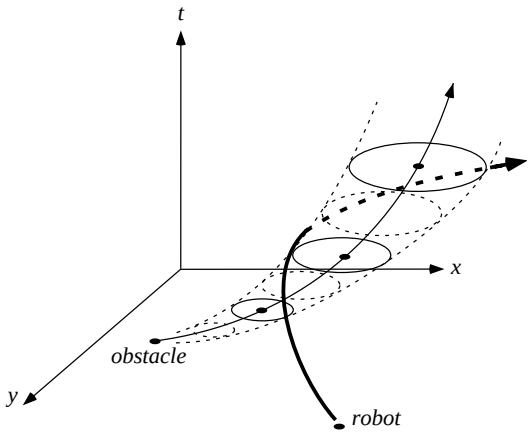


Figure 8. Path planning in time-space.

4.3.1 Static Model Selector

This is a very simple frequency-based selector which does not consider much about the dynamics of the environment. Let M_i be models and P_i be their probabilities. Each probability is calculated from the relative frequency in the latest N_h trials.[1] Let n_i be the frequency of the ith model in the trials. The probability P_i is given by

$$P_i = \frac{n_i}{N_h}.$$

Let i^* be the index of the most probable model. If P_{i^*} is above a predetermined threshold, the i^*th model is selected; otherwise, the model selector considers that the obstacle does not have no canonical trajectory and considers all possible moving directions of the obstacle derived from all possible candidates for the canonical trajectory. These two cases are analogous to the case in Section 2 where the situation is known to be J_1 or J_2 and the case where the situation is unknown, respectively.

[1] We assume that, after each trial, the robot can uniquely determine the motion model of the obstacle during the trial.

4.3.2 Dynamic Model Selector

This is also a simple frequency-based selector, but it investigates the underlyning dynamics of the environment (i.e., moving obstacle). That is, the selector considers the change of the moving obstacle's destination as a Markov process and estimates the transition matrix M of the Markov process from the history of obstacle motion. The element $M_{i,j}$ of the matrix indicates the probability that the obstacle that took the ith model at the latest trial takes the jth model at the next trial, and is estimated by:

$$M_{i,j} = \frac{n_{i,j}}{N_i}, \tag{1}$$

where $n_{i,j}$ is the frequency of the jth model taken just after the ith model and N_i is the frequency of the ith model.

4.4 Simulation Results

Figure 9 shows the problem setting for simulation. The robot moves upward and the obstacle moves downward. There are three motion models of the obstacle, which are referred to as LW (leftward, from the robot's point of view), ST (straight), and RW (rightward). A canonical trajectory of the obstacle is calculated for each pair of the current and the goal position. The robot classifies the situation into one of these three models.

In the simulation, we can consider two kinds of motion models of the obstacle: one is the model that the robot expects for the obstacle (*expected model*); the other is the model that the obstacle actually takes (*actual model*). If these two models coincide, the robot motion is expected to be efficient; otherwise, the robot is likely to exhibit an inefficient behavior. Figure 10 shows two exmaples of trials. In Figure 10(a), the robot thought the obstacle would move leftward (LW) and the obstacle actually moved as expected (LW); in the Figure 10(b), on the other hand, although the robot thought the obstacle would move rightward (RW), the obstacle actually moved leftward (LW). The robot motion is much more efficient in the first case than in the second.

Table 3 shows the result of simulation trials to see how the relationship between the expected and the actual models affects to the performance of the robot. In the table, each

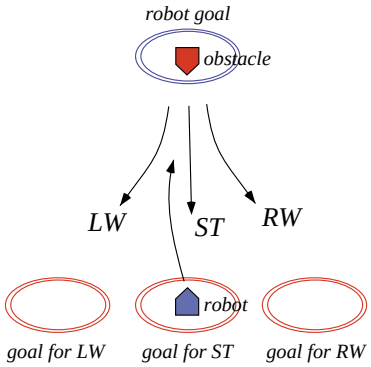
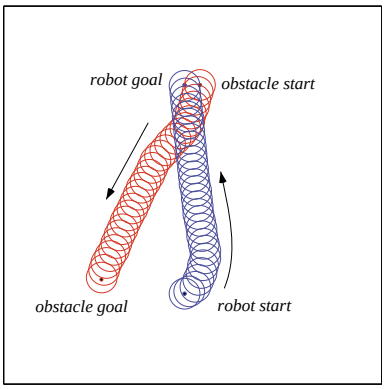
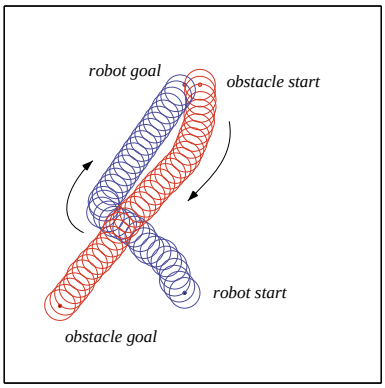


Figure 9. Problem setting.



(a) expected=LW, actual=LW.



(a) expected=LW, actual=RW.

Figure 10. Results of two trials.

Table 3. Simulation results.

		ACTUAL			
		NO	ST	LW	RW
E	NO	32.55	34.59	33.18	33.31
X		(6.57)	(2.58)	(4.39)	(5.05)
P	ST	38.21	29.63	31.26	35.34
E		(12.14)	(1.02)	(3.49)	(6.82)
C	LW	37.01	31.35	29.72	34.39
T		(8.49)	(1.81)	(2.34)	(5.71)
E	RW	32.97	31.41	32.89	29.71
D		(6.16)	(1.82)	(2.73)	(2.62)

row indicates the expected model and each column indicates the actual model. The NO-row indicates that the robot has no models of obstacle motion; the NO-column indicates that the obstacle randomly selects one of the three models (LW, ST, RW). For each combination of the two models, we ran 150 trials. In each box, the upper number is the mean of the time steps the robot took to reach the destination. The lower number in parentheses is the standard deviation of the time steps. We can see from the table that the robot motion (and equivalently the planning result) is efficient if an appropriate planning model is selected, and it is not otherwise.

Next we tested the three-level planning architecture through a large number of concecutive trials. The knowledge-based

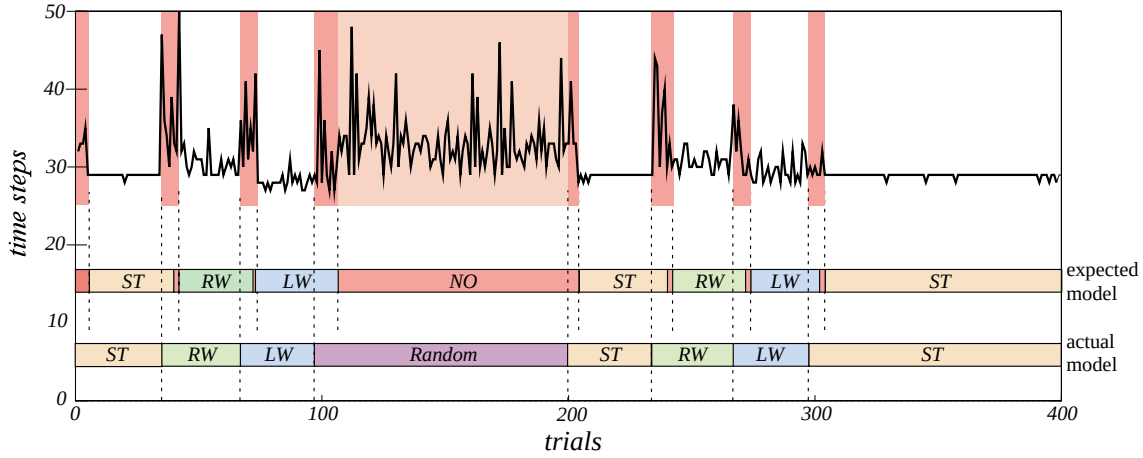


Figure 11. obstacle's actual behavior, expected behavior, and elapsed time steps.

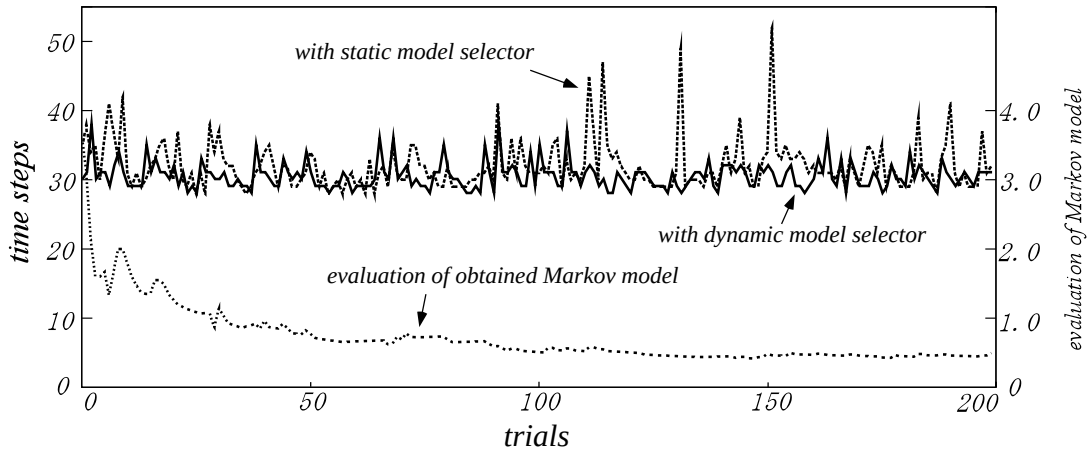


Figure 13. Comparison of planners with dynamic and static model selector.

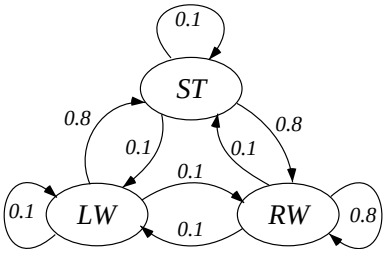


Figure 12. Markov process used.

model selector collects the history of the obstacle movement and determines the current model based on the probability estimation method mentioned above.

The first test was done with the static model selector (see Sec. 4.3.1). Figure 11 shows the result. The figure shows the time elapsed by the robot to reach the destination (upper part), the expected model of the obstacle (middle part), and the actual model of the obstacle (lower part). In the upper part, the ranges of trials where there was a discrepancy between the expected and the actual model are shown as shaded regions. For the NO actual model, the NO is the best expected model, but in this case, we can say that there is always a discrepancy; so the range corresponding to such a case is also lightly shaded. In such shaded regions, the robot motion is not efficient due to generating a plan based on an inappropriate planning model.

The second test was done with the dynamic model selector (see Sec. 4.3.2). In this case, we used the Markov process shown in Figure 12 as the underlyning dynamics to generate the problem sequence. In addition, the *expected* motion model

is selected so that the predicted expectation of the cost is minimized using the predicted probability of each model and also using Table 3 as the expected cost for each combination of the actual and the predicted model. Figure 13 shows the result. In the figure, the performance of the planning system with the static model selector and that with the dynamic model selector as well as the change of the evaluation of the estimated Markov model[2]in the dynamic model selector are shown. The result shows the dynamic model selector outperforms the static one because the dynamic model selector utilizes the knowledge about the problem structure that the obstacle changes the motion model according to some Markov process.

5 Example Domain 2: Intelligen t Driver Assistance

This section briefly describes an application of the three-level planning architecture to the *intelligent navigator* that can give the driver timely advice on safe and efficient driving. For the details of the intelligent navigator, refer to [12].

Usually tasks in driving can be divided into two levels [17]: the *tactical level* determines maneuvers such as lane changing and overtaking to meet the objective of driving (e.g., a target arrival time) under the constraints imposed by the actual traffic condition; the *operational level* translates the selected maneuver into actual operations of steering, accelerating, adn braking.

In real traffic, sensory data based on which the intelligent navigator generates advice is *uncertain* (e.g., measurement error or occlusion). In addition, the situation is *dynamic*, i.e., the situation evolves as time elapses. Therefore, the tactical level advice generation should be based on the prediction of the future traffic condition with consideration of uncertainty. We adopt a decision-theoretic planning for the tactical level.

Then, in order to adaptively select a planning model for the tactical level and to activate the tactical level only when it is necessary, we introduce a meta-level planning (called the *meta-tactical level*). The resultant control architecture is composed of three levels: *meta-tactical level*, *tactical level*, and *operational level*; they exactly correspond to the three levels shown in Figure 4.

5.1 Tactical Level Decision-Thoretic Planning

Figure 14 shows a scenario where a decision-theoretic planning is employed to determine a maneuver. Our vehicle (painted rectangle) is on the left lane [3], and is approaching the exit to take. Since the speed in the current lane is becoming a little bit slow, the driver starts thinking of overtaking vehicles ahead. The overtaking maneuver is generally faster, but there may be risks of lane changing itself and of missing the exit due to occluded vehicles ahead. Such a trade-off is formalized in a *planning model* corrsponding to each situation.

In this application, each planning model is composed of the following:

[2] The model is evaluated by the sum of absolute differences of the corresponding elements of the obtained transition matrix and the true matrix.

[3] Note that the slower lane is the left one in Japan.

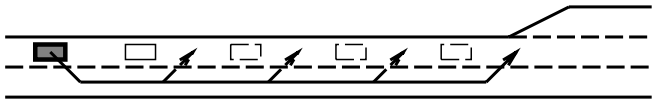


Figure 14. An overtaking scenario.

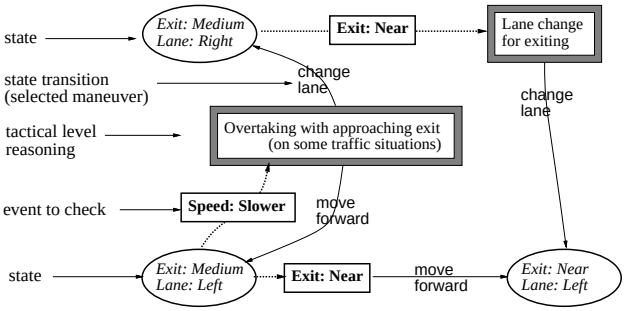


Figure 15. A part of state-transition graph for the meta-tactical level.

- a state is represented by the position and the velocity of our vehicle and those of other vehicles.
- an action is a maneuver. In the above scenario, there are two actions: *change lane for overtaking* and *go straight*.
- a probability distribution is calculated over the possible future placements of vehicles including occluded ones.
- a utility function which is a function of the expected time of our vehicle reaching the exit and the loss of each maneuver.

We also have constructed planning models for other traffic situations such as: overtaking near the target exit without congestion; congestion around an entrance or an exit which our vehicle does not take. All such models are manually constrcuted.

5.2 Meta-T actical Lev el as Knowledge-Based Model Selector

This level continusouly watches important events on traffic. Examples of possible events are: the average speed of the current lane slows down; the exit is approaching. It also periodically updates the estimation of the arrival time. Since it is inefficient to always check all events, only selected events are monitored which are considered to be important in the current state. To realize such an adaptive focus of attention, we construct a state transition graph. Figure 15 shows a part of the transition graph. For example, at state `[Exit: Medium, Lane: Left]` (which means that the distance to the exit is medium and the vehicle is on the left lane), possible events are: (1) the speed becomes slower (`Speed: Slower`); (2) the exit becomes near (`Exit: Near`). For each event, the corresponding planning model for the tactical level is assigned. This strcuture enables the meta-tactical level to adaptively select appropriate planning models.

5.3 Implemen tation and Experimen ts

The advice generation subsystem with the three-level architecture is connected to a road scene visual recognition subsystem, which detects and localizes lanes and other vehicles, to constitute the intelligent navigator. We constructed a prototype system and conducted experiments on an actual high way. In one case, for example, our vehicle with the intelligent navigator traveled about 12km, and during the travel, the intelligent navigator generated advice 5 times, all on appropriate timings.

6 Conclusions and Discussion

This paper has discussed the importance of analyzing the problem structure and of the model selection mechanism for an agent making a plan in uncertain and dynamic environments. A three-level planning architecture has been proposed which has a model selection layer on top of an decision-theoretic middle layer. We applied the architecture to two planning problems. In a mobile robot planning in a dynamic environment, the model selector is implemented as a frequency-based model estimator, which can select an appropriate planning model adaptively. Model selection by the dynamic model selector is done based on the expected loss due to model discrepancy. In the intelligent navigator example, the model selector is implemented as a state-transition graph, which selects planning models according to both the history of maneuvers and the current traffic condition. These two application examples show the generality and the usefulness of the proposed architecture.

This paper has focused only on the use of the knowledge-based meta-level control for planning model selection. Another important role of the meta-level control is to limit the search space adaptively according to the time pressure. The following two approaches are possible: limiting the set of action candidates and limiting the length of lookahead. These kinds of meta-level control could be realized by a method which explicitly considers the tradeoff between the amount of search and the plan quality. However, formulating such a tradeoff could sometimes be difficult in complex planning problems in the real world. Therefore, our approach of employing knowledge-based control seems to have an advantage in a practical use.

In this paper, we have manually constructed the meta-level model selector by examining the problem structure. A future work is to automate the construction of the model selector to some extent, by using various machine learning techniques.

REFERENCES

[1] A.G. Barto, S.J. Bradtke, and S.P. Singh, 'Learning to act using real-time dynamic programming', *Artificial Intelligence*, **72**(1-2), 81–138, (1995).

[2] J. Blythe, 'Decision-theoretic planning', *AI Magazine*, **20**(2), 37–54, (1999).

[3] M. Boddy and T. Dean, 'Solving time-dependent planning problems', in *Proceedings of IJCAI-89*, pp. 979–984, (1989).

[4] C. Boutilier, R. Dearden, and M. Goldszmidt, 'Exploiting structure in policy construction', in *Proceedings of the Fourteenth Int. Joint Conf. on Artificial Intelligence*, pp. 1104–1111, (1995).

[5] W.L. Buntine, 'Operations for learning with graphical models', *J. of Artificial Intelligence Research*, **2**, 159–225, (1994).

[6] A. Cameron and H. Durrant-Whyte, 'A bayesian approach to optimal sensor placement', *Int. J. of Robotics Res.*, **9**, 70–88, (1990).

[7] T. Dean, L.P. Kaelbling, J. Kirman, and A. Nicholson, 'Planning with deadlines in stochastic domain', in *Proceedings of AAAI-93*, pp. 574–579, (1993).

[8] B. Pell et al., 'An autonomous spacecraft agent prototype', in *Proceedings of Agents-97*, (1997).

[9] E. Gat, 'Integrating planning and reacting in a heterogeneous asynchronous architecture for controlling real-world mobile robots', in *Proceedings of AAAI-92*, pp. 809–815, (1992).

[10] E.J. Horvitz, *Computation and Action Under Bounded Resources*, Ph.D. dissertation, Stanford University, 1990.

[11] S.A. Hutchinson and A.C. Kak, 'Planning sensing strategies in a robot work cell with multi-sensor capabilities', *IEEE Trans. on Robotics and Automat.*, **5**(6), 765–783, (1989).

[12] M. Itoh, J. Miura, and Y. Shirai, 'Towards intelligent navigator that can provide timely advice on safe and efficient driving', in *Proceedings of the 1999 IEEE/IEEJ/JSAI Int. Conf. on Intelligent Transportation Systems*, pp. 981–986, Tokyo, Japan, (October 1999).

[13] J. Miura and Y. Shirai, 'Vision and motion planning for a mobile robot under uncertainty', *Int. J. of Robotics Research*, **16**(6), 806–825, (1997).

[14] J. Miura and Y. Shirai, 'Vision-motion planning for a mobile robot considering vision uncertainty and planning cost', in *Proceedings of the 15th Int. Joint Conf. on Artificial Intelligence*, pp. 1194–1200, Nagoya, Japan, (August 1997).

[15] S. Russell and E. Wefald, *Do The Right Thing*, The MIT Press, 1991.

[16] N. Suematsu and A. Hayashi, 'Consideration of model uncertainty in decision theoretic planning', *Trans. Information Processing Soc. of Japan*, **40**(7), (1999). (in Japanese).

[17] R. Sukthankar, S. Baluja, J. Hancock, D. Pomerleau, and C. Thorpe, 'Adaptive intelligent vehicle modules for tactical driving', in *1996 AAAI Workshop on Intelligent A daptive Agents*, pp. 13–22, (1996).

4SP: A four-stage incremental planning approach[1]

Eva Onaindia, Laura Sebastia and **Eliseo Marzal** [2]

Abstract. GraphPlan-like and SATPLAN-like planners have shown to outperform classical planners for most of the classical planning domains. However, these two propositional approaches do not exhibit good results for large-size problems due to the graph size they have to handle.

In this paper we show a new approach for planning as an attempt to combine the advantanges of a graph-based analysis and a partial-order planner. The fast responses obtained in the experimental results show that the application of this technique can report significant benefits in terms of a reduction in search space and that the average performance of this planner is much better: it takes a bit longer to solve some easy problems but it is capable to easily solve large problems.

1 INTRODUCTION

Recently, a study to compare the performance and limits of six planners reports that, to date, no one planner has demonstrated clearly superior performance [6]. The conclusion of the study is that the best planner varies accross problems. The planners were tested on the UCPOP suite problems and on a particular software testing domain from the authors [6]. No planner solved all of the problems. Some planners as STAN [4] were faster in general and others solved a few more problems the others did not. As for the software testing domain, UCPOP [1] clearly dominated and solved many more problems than the others.

Moreover, we tested STAN [4] and Blackbox [5] on large problems from the blocks world domain and noticed that none of them were able to solve problems involving more than fifteen blocks.

Our motivation is to develop a new planning approach which also offers a good performance for large size problems. In order to tackle this issue, we present a planner which integrates a graph-based preprocessing technique that incrementally exploits the problem knowledge and a partial-order planner (POP).

Our new planning approach, 4SP, executes a four-stage algorithm:

- First stage: its task is to build up a graph that will contain the set of all of the possible actions which can be executed at each time point.
- Second stage: the result of this phase is a more refined graph which only contains a subset of actions that must necessarily occur in a correct solution.
- Third stage: the purpose of this stage is to guarantee a partial consistency between the actions in the graph. The graph obtained from this phase will even comprise, in some particular cases, all of the actions of a correct solution.
- Fourth stage: this stage is aimed at adding the missing actions in the plan and finding an ordering relation for all the actions in the plan (total consistency). The result of this phase will be the final solution plan.

The experimental results show that a proper integration of a graph-based preprocessing technique and a POP reports significant benefits in terms of a reduction in search space and it is also capable to solve large size problems. The layout of the paper is as follows: in sections 2 through 5 we will focus on each planning stage, section 6 shows the obtained results and section 7 draws conclusions from this work and summarizes some directions for future work.

2 THE FIRST STAGE

The first phase of the algorithm creates a graph inspired in a Graphplan-like expansion. This graph, named *problem graph*, may partially or totally encode the problem. The problem graph is a directed, layered graph with two kinds of nodes (literals and actions) and two kinds of edges (precondition-edges and add-edges). The levels alternate action levels containing action nodes and literal levels containing literal nodes.

- An action-level A_j consists of all of the action instantiations which are applicable in the previous literal-level L_{j-1} and are different from any other action instatiation in the graph. That is, A_j is composed of all of the action instantiations a_{jk} which satisfy these two requiments:
  - all preconditions of a_{jk} are present in the previous literal-level L_{j-1} and
  - a_{jk} does not occur in any previous action level
- A literal level L_j is a set of propositions implictly representing the different world states reachable after executing actions in A_j. More specifically, let $A_j = \{a_{j1}, a_{j2}, \ldots, a_{jn}\}$ be the set of action instantiations that can be executed at action level A_j; the set of literals in L_j is defined as $L_{j-1} \cup \mathsf{AddEff}(a)$[3] $\forall a \in A_j$, that is literals in the previous level L_{j-1} plus the add effects of each action in A_j.

The first level in the problem graph is the literal-level L_0 and it is formed by all literals in the initial situation. A_1 consists of all of the action instantiations which are applicable in L_0. L_1 is the set of literals in L_0 plus the add effects of each action in A_1 and so forth. The problem graph creation terminates when a literal level containing all of the literals from the goal situation is reached in the graph or when no more new actions can be applied. Notice that the delete

[1] This work has been partially funded by the Spanish Government CICYT-FEDER project 1FD7-0887

[2] Dept. Sistemas Informaticos y Computacion, Technical University of Valencia, 46071 Valencia, Spain, email: {onaindia, lstarin, emarzal}@dsic.upv.es

[3] AddEff, DelEff and Pre stand for the add effects, delete effects and preconditions of an action respectively.

effects of actions are ignored during the problem graph creation and therefore interactions between actions are not taken into account at this stage. This makes the first stage be a very fast polynomial time process.

It must also be noticed that a problem graph is neither a state-space graph nor a Planning Graph [2]. There are two main differences with respect to a Planning Graph:

a) levels in the problem graph do not stand for time steps but for instantiation steps which can entail more than one execution step. An action level A_j denotes that every action in A_j will be executed at a time step $t \geq j$ and at least one action from A_{j-1} must be executed firstly.
b) since delete effects of actions are ignored in the problem graph we do not have to deal with mutual exclusion relations among nodes at this stage. The next phase will be responsible for identifying the relation between two actions in the same level: mutually exclusive (they interfere with each other), complementary actions (one of the actions adds an effect the other needs) or independent actions (there is no explicit order between them).

To illustrate the process of the problem graph creation we will take the *hanoi* problem for three disks (big -B-, medium -M- and small -S-) and three pegs (P1, P2 and P3) as an example (Table 1). The first step is to build the literal level L_0 which comprises all of the literals in the initial situation (literals are numbered as they appear in the graph). Following, the action level A_1 is created by finding all possible applications of the operator Move ?disk ?place1 ?place2 over the literals in L_0, where ?place1 and ?place2 may indicate a disk or a peg. Once the first action level is created, the next literal-level, L_1, will include the set of literals in L_0 plus the new effects added by the two actions in A_1. In Table 1 literals 2 and 3 are in bold to indicate they also belong to the goal state. The column next to A_1 shows the preconditions required by each action and the add effects generated by the action (P stands for preconditions and E stands for add effects).

L_0		A_1		L_1	
B on P1	1	Move S M P2	P={3,4,5}, E={7,8}	B on P1	1
M on B	2	Move S M P3	P={3,4,6}, E={8,9}	**M on B**	2
S on M	3			**S on M**	3
clear S	4			clear S	4
clear P2	5			clear P2	5
clear P3	6			clear P3	6
				S on P2	7
				clear M	8
				S on P3	9

Table 1. Hanoi problem graph (1)

Next step is to generate the actions in A_2 by applying the operator Move ?disk ?place1 ?place2 over the literals in L_1. Only the instantiations which have not been inserted in the graph yet (that is, instantiations different from Move S M P2 and Move S M P3) are considered. In order to do this, we only take into account those instantiations which involve at least one of the new literals inserted at L_1 (7, 8 or 9), as the rest of instantiations already appear at the previous action-level A_1. The second level of actions and literals (A_2 and L_2) are shown in Table 2. The new literals are those numbered from 10 to 12.

A_2		L_2	
Move S P2 M	P={4,7,8}, E={3,5}	B on P1	1
Move M B P2	P={2,5,8}, E={10,11}	**M on B**	**2**
Move M B P3	P={2,6,8}, E={10,12}	**S on M**	**3**
Move S P3 M	P={4,8,9}, E={3,6}	clear S	4
Move S P2 P3	P={4,6,7}, E={5,9}	clear P2	5
Move S P3 P2	P={4,5,9}, E={6,7}	clear P3	6
		S on P2	7
		clear M	8
		S on P3	9
		clear B	10
		M on P2	11
		M on P3	12

Table 2. Hanoi problem graph (2)

Step 3 follows the same rules explained above to create A_3 and L_3. All of the literals in the goal situation finally appear at L_3 (Table 3) and consequently the process of the problem graph creation finishes.

A_3		L_3	
Move B P1 P2	P={1,5,10}, E={13,14}	B on P1	1
Move B P1 P3	P={1,6,10}, E={14,16}	**M on B**	**2**
Move M P2 B	P={8,10,11}, E={2,5}	**S on M**	**3**
Move M P3 B	P={8,10,12}, E={2,6}	clear S	4
Move M P2 P3	P={6,8,11}, E={5,12}	clear P2	5
Move M P3 P2	P={5,8,12}, E={6,11}	clear P3	6
Move S P2 B	P={4,7,10}, E={5,15}	S P2 B	7
Move S P3 B	P={4,9,10}, E={6,15}	clear M	8
Move S M B	P={3,4,10}, E={8,15}	S on P3	9
		clear B	10
		M on P2	11
		M on P3	12
		B on P2	13
		clear P1	14
		S on B	15
		B on P3	**16**

Table 3. Hanoi problem graph (3)

The problem graph may include all of the actions of a solution plan. For all of the tested domains (see section Experimental Results), except the *hanoi* problem, we obtained a problem graph which

includes all necessary actions in a valid solution plan. However, this cannot be guaranteed because, as it was said above, the problem graph creation finishes at a level where all the literals from the goal situation are present, even though additional actions could be applied at this final level.

The advantage of the problem graph is that its size is much smaller than the Planning Graph and the cost of creating this graph is hardly appreciable even when dealing with large size problems.

3 THE SECOND STAGE

The goal of this phase is to extract the information concerning the planning problem from the problem graph. At this stage, a new graph, named *basic graph*, is obtained. The basic graph is created by selecting from the problem graph only those actions which must necessarily appear in a valid solution.

The basic graph is a directed, layered graph with only action nodes. The number of levels in the basic graph is the number of action levels in the problem graph plus two additional levels, an initial and a final action level. The former contains one action a_0 which has effects and no preconditions; the final level includes one action a_n with preconditions and no effects.

The process to create the basic graph starts with preconditions of a_n (goal literals). The objective is to find a set of actions in the problem graph having these goals as add effects. The found actions are inserted in the basic graph and their preconditions form a new set of subgoals which in turn are solved by following the same process. Once there are some actions in the basic graph, the new preconditions can be achieved with actions from the problem graph or basic graph. At the end of this phase the basic graph will be a subset of the actions in the problem graph which belong to a correct solution.

In order to find the correct action for each literal (subgoal), 4SP applies the following property:

Property 1 (literal consistency) *A literal p required by an action a_k ($p \in \mathsf{Pre}(a_k)$) is said to be consistent if these two requirements hold:*

- *there is a sequence of actions $a_i \rightarrow a_{i+1} \ldots a_{k-1} \rightarrow a_k$ such that $p \in \mathsf{AddEff}(a_i)$ and $p \notin \mathsf{DelEff}(a_j)\ \forall j \in [i+1, k-1]$.*
- *for each action a_i such that $p \in \mathsf{DelEff}(a_i)$ there is a sequence $a_i \rightarrow a_{i+1} \ldots a_{k-1} \rightarrow a_k$ with an action a_j, $j \in [i+1, k-1]$, such that $p \in \mathsf{AddEff}(a_j)$.*

In order to check the literal consistency it is necessary to propagate effects of an action a_i each time a causal link $a_i \rightarrow a_j$ is asserted. The propagated effects of an action a_j are computed by means of the following procedure:

1. $\mathsf{PDelEff}(a_0) = \mathsf{DelEff}(a_0)$
   $\mathsf{PAddEff}(a_0) = \mathsf{AddEff}(a_0)$
2. Let $p_0, p_1, \ldots, p_n$ be the paths in the graph that have a_j as destination node. Let $A = \{a_{0,j-1}, a_{1,j-1}, \ldots, a_{n,j-1}\}$ be the set of predecessor actions of a_j, each corresponding to a path.

   (a) $\mathsf{PAddEff}(a_j) = \{x \in \mathsf{PAddEff}(a_i) : a_i \in A / (\exists a_k \in A \wedge x \in \mathsf{PDelEff}(a_k)) \rightarrow a_k < a_i\}$ [4]

   $\mathsf{PDelEff}(a_j) = \{x \in \mathsf{PDelEff}(a_i) : a_i \in A / (\exists a_k \in A \wedge x \in \mathsf{PAddEff}(a_k)) \rightarrow a_k < a_i\}$

   (b) $\mathsf{PAddEff}(a_j) = \mathsf{PAddEff}(a_j) - \mathsf{DelEff}(a_j)) \cup \mathsf{AddEff}(a_j)$
   $\mathsf{PDelEff}(a_j) = \mathsf{PDelEff}(a_i) - \mathsf{AddEff}(a_j)) \cup \mathsf{DelEff}(a_j)$

[4] $a_k < a_i$ denotes action a_k is executed before action a_i

3.1 Algorithm for the basic graph

The aim of this second phase is to obtain a basic graph where the property of literal consistency is satisfied for each action precondition. In order to check whether a literal is consistent or not the delete effects of the producer action of a causal link must be propagated according to the procedure stated in section 3. Figure 1 shows some cases of inconsistent literals. In Figure 1.a, action a_1, which is selected to satisfy the precondition y of a_2, provokes a conflict as it deletes the precondition x of a_3. Figure 1.b shows a similar example where the action a_1 which is introduced in the basic graph to satify a precondition also deletes a literal of the same needer action.

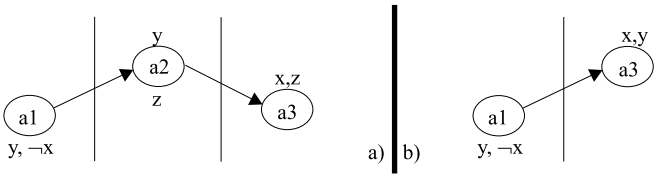


Figure 1. Some examples of inconsistent literals

The key point of the algorithm for the basic graph is to select the proper actions to satisfy the preconditions of the actions inserted in the basic graph. In order to select an action a_i from the problem graph to solve a precondition p of an action $a_j \in A_j$ in the basic graph, the algorithm proceeds as follows:

1. **Find a set of actions for** p. Find all actions at any level $A_i \leq A_j$ having p as an add effect.
   - If the actions found in the problem graph belong to different action levels we say p is an OPEN literal. In this case a set R containing all the different levels the literal belongs to is created. When a literal is OPEN it is not possible to make a decision about which action to choose and the literal remains as OPEN until more information is available.
   - If all actions found in the problem graph belong to the same action level we say p is a KNOWN precondition. In this case it is not possible yet to choose the proper action to satisfy p but at least the level of the producer action is known. All of the actions which produce p are gathered in a cluster and the common preconditions, add and delete effects of the cluster are identified. From this point, the cluster is treated as a single action (with its preconditions, add effects and delete effects) until one of the actions in the cluster is selected.
   - If there exists only one action to satisfy p then the action is clearly identified and it is inserted in the basic graph. In this case p is a CLOSED literal.

   From a set of literals to be solved, 4SP selects first CLOSED literals, then KNOWN literals and finally OPEN literals.
2. **Study the delete effects of the selected actions**. If p is a KNOWN or CLOSED literal then one of the actions in the cluster, or the selected action, must necessarily be used to solve the precondition p.

4SP analyzes the consequences of propagating the common delete effects of the actions in the cluster or the delete effects of the selected action respectively. The aim of the propagation is to find out whether any other literal in the basic graph becomes inconsistent after adding the causal link $a_i \rightarrow a_j$ for p ($a_i \in A_i$ will be the selected action if p is a CLOSED literal or one of the actions in the cluster if p is a KNOWN literal). Let's suppose that a precondition q of an action a_k in the basic graph ($a_k \in A_k, A_k \geq A_i$) becomes inconsistent after propagating the delete effects of a_i:

(a) If q is a CLOSED or KNOWN literal, an ordering between a_i and the producer action of q for a_k is added.

(b) If q is an OPEN literal, the new set of action levels for q is computed as $R = \{A_h\}/h \in [i+1, k]$, and so only actions in an action level of R are now considered as potential producers for q. In short, the range of action levels for a precondition q is restricted by discarding all levels lower and equal than the action level which deletes q. In both cases of Figure 1 literal x of action $a3$ becomes inconsistent due to the propagation of the delete effects of action $a1$. In figure 1-a), x could be achieved with actions in the same level as $a3$ or from any lower level, whereas in figure 1-b) x could only be achieved with actions from its own level.

3. **Select or limit the number of actions for** p.

(a) When p is an OPEN literal the producer action for p is not known, not even the action level for that action. As it was explained above, the number of action levels (R) of a literal can be restricted as long as new information is inserted in the basic graph. In this way, it may eventually happen that $|R| = 1$ and thus the literal becomes KNOWN or CLOSED. Otherwise, when all CLOSED and KNOWN literals have been studied (at this point one iteration of the algorithm is completed), the upper action level of all OPEN literals is removed. Actions from lower levels are preferred because the lower the level of the action is the less actions will have to be introduced to achieve its preconditions. The behaviour of the algorithm always follows a "principle of minimality" which is also applied at other points. This will be explained later on.

(b) If p is a KNOWN literal, and therefore there is a potential set of actions to achieve p, the algorithm discards those actions for which property 1 does not hold. That is, $\forall a_i/p \in \mathsf{AddEff}(a_i)$, if the causal link $a_i \rightarrow a_j$ violates property 1 for any other literal in the basic graph then a_i is removed from the set of actions. Sometimes the application of this property is not sufficient to discriminate among a set of actions and consequently additional criteria are to be applied.

The algorithm selects firstly actions in the basic graph than in the problem graph when achieving a precondition of an action. The reason is that it is preferable to use actions already existing in the basic graph than adding a new action (given two literals p and q to be satisfied, if an action $a1$ in the problem graph achieves p and an action $a2$ in the basic graph achieves both p and q then $a2$ will be selected). On the other hand, when there are several potential producer actions for a precondition, and all of them satisfy property 1, the algorithm selects the one which has less preconditions unresolved. The application of these two criteria indicate the process for creating the basic graph is oriented towards obtaining the minimal set of actions. In case none of the them allows to discriminate among the actions, any action will be valid.

3.2 Properties of the basic graph

After the second stage two different results can be obtained:

- **No basic graph exists**. This happens when it is not possible to find a sequence of actions to achieve a particular literal, and it is usually due to the lack of an operator to achieve such a literal. Let's take the example in Figure 2 where the action a_i deletes the precondition p of action a_k. In case there is no operator for achieving p, the only way to satisfy the literal is with the effects of the initial situation and therefore the precondition p of the action a_k will never be a consistent literal. Notice that the action a_k appears in the problem graph (and consequently in the basic graph) because at the literal-level L_j both p and q are present and q is a new literal achieved in the previous action-level A_j. If no basic graph is obtained the problem is unsolvable.

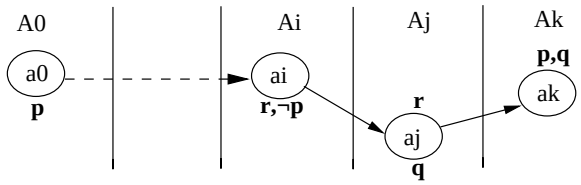


Figure 2. No basic graph

- **A basic graph is obtained**. Although the basic graph is created, this does not guarantee the problem is solvable. That is, the fulfilment of property 1 is not sufficient to discover unsolvable problems. However, if a solution exists for a problem, all of the actions in the basic graph will be part of such a solution.
  Basically, the basic graph comprises optimal sequences of actions to achieve each subgoal literal independently. Only a few interactions among actions are considered at this stage, as those due to the introduction of causal links. For this reason, in most cases it is not possible to establish a set of consistent ordering constraints among all of the actions in the basic graph.

The issues of completeness, soundness and termination on unsolvable problems are tackled in section 4.

3.3 An application example

In order to show the process for creating a basic graph we will take the example of the *hanoi* problem. The starting point is the problem graph shown in Tables 1, 2 and 3.

The algorithm begins with the literals in the final situation: 2, 3 and 16. Literals 2 and 3 are OPEN because literal 2 can be achieved with the initial action a_0, as it is one of the initial effects, and actions at level A_3; literal 3 is also produced by action a_0 and two actions at level A_2. The algorithm selects literal 16 (CLOSED) and action Move B P1 P3 is inserted in the basic graph at $A3$. The preconditions of the new actions are 1, 6 and 10. Literal 1 is CLOSED (level A_0), literal 6 is OPEN (levels A_0, A_2 and A_3) and literal 10 is KNOWN (two actions at level A_2). Since literal 1 is a CLOSED literal (as it only appears at A_0), a new causal link between a_0 and Move B P1 P3 is added in the basic graph (Figure 3).

Next step is to study literal 10. There are two choices at A_2 for literal 10 (Move M B P3 and Move M B P2). Both actions have a

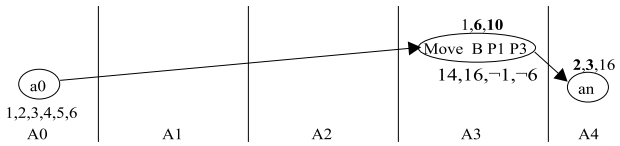


Figure 3. Basic graph 1 for the hanoi problem

common delete effect, literal 2, which makes precondition 2 of action a_n be an inconsistent literal (after applying the effects propagation).

At least one of these two actions must be chosen to solve literal 10; precondition 2 of a_n is an OPEN literal so the number of action levels for literal 2 is restricted to $\{A_3\}$. A cluster with both actions for literal 10 is created (Figure 4). We discard Move M B P3 because this action gives rise to a literal inconsistency as it deletes literal 6 which is a precondition for the action Move B P1 P3, whereas Move M B P2 does not cause any conflict.

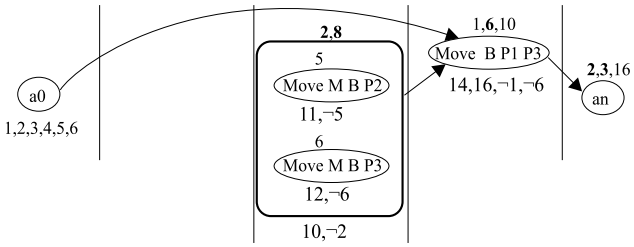


Figure 4. Basic graph 2 for the hanoi problem

After some more steps, the situation is as shown in Figure 5. At this point, we have no criteria to choose between the actions in the clusters.

1. At least one of the two actions in the cluster at A_1 must be chosen to generate literal 8. Both actions produce a literal inconsistency (Move S M P2 deletes precondition 5 of action Move M B P2 and action Move S M P3 deletes precondition 6 of action Move B P1 P3) and we cannot discriminate between them by any other criteria.

   (a) If we tried to solve the conflict generated by Move S M P2 then literal 5 would have to be achieved for action Move M B P2; there are two actions that have literal 5 as an add effect (Move S P2 M in the basic graph and Move S P2 P3 in the problem graph) but both generate a literal inconsistency (the former deletes literal 8 which is a precondition of Move M B P2 and the latter deletes literal 6 which is a precondition of Move B P1 P3).

   (b) If we tried to solve the conflict generated by Move S M P3 then literal 6 would have to be achieved for action Move B P1 P3; there are two actions having literal 6 as an add effect (Move S P3 M in the basic graph and Move S P3 P2 in the problem graph); as the former is in the basic graph and it does not cause any conflict we conclude that the correct choice to solve literal 8 is by selecting the action Move S M P3.

2. Once the action in the cluster at $A1$ is known we proceed with the cluster at $A2$. Since we have selected Move S M P3 at A_1, it is easy to see that Move S P3 M is the correct action at A_2 because all of its preconditions are already solved whereas Move S P2 M would need an additional action to achieve its precondition 7.
3. The same criteria can be applied to the cluster at A_3, thus resulting in the selection of Move M P2 B as all of its preconditions are already solved with actions in the basic graph.

At this point the upper action levels for literals 5 and 6 are removed. Literal 5 is then solved with action a_0 and literal 6 with the action selected at $A2$, Move S P3 M.

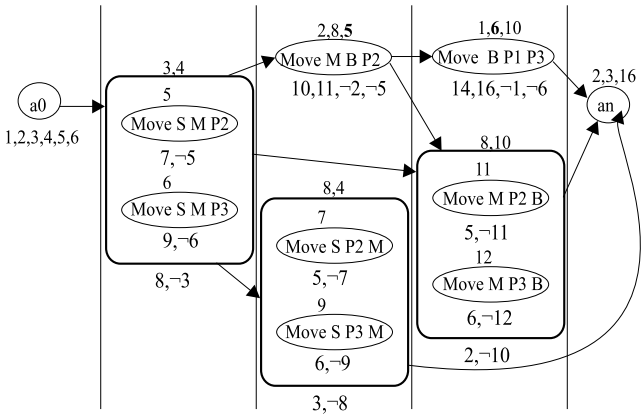


Figure 5. Basic graph 3 for the hanoi problem

The resulting basic graph (Figure 6) comprises five out of the seven necessary actions to solve the problem.

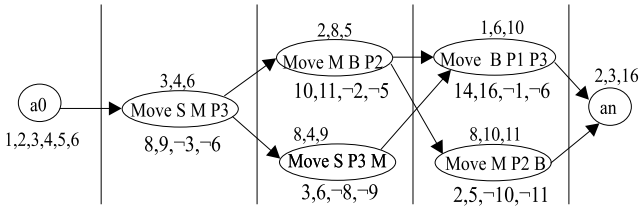


Figure 6. Final basic graph for the hanoi problem.

4 THE THIRD STAGE

Property 1 guarantess that, if a solution exists for the problem, all of the actions in the basic graph will belong to a correct solution. However, having obtained a basic graph is not a sufficient condition to ensure the problem is solvable. On the other hand, even though the problem is solvable, it may be impossible to set an order among

all of the actions in the basic graph. In short, the set of actions in the basic graph constitute a first approximation towards a final plan but still some further refinements can be done on the graph.

The third stage is aimed at solving these issues and obtaining a more refined graph. The behaviour of the third stage is guided by the following property:

Definition 1 (partial consistency) *A basic graph is said to be partially consistent if it is possible to set a total-order relation between each pair of actions in the same action level of the basic graph.*

The application of this property will enable:

1. to discover unsolvable problems,
2. to get a more refined graph, closer to a final solution plan,
3. to provide a support towards obtaining an optimal solution (the shortest solution, i.e. the one with the less number of actions).

Let a_1 and a_2 be two actions of a same level action, $\mathsf{Pre}(a_1) = \{x_1, y\}$, $\mathsf{Eff}(a_1) = \{z_1, \neg y\}$, $\mathsf{Pre}(a_2) = \{x_2, y\}$, $\mathsf{Eff}(a_2) = \{z_2, \neg y\}$. Clearly, it is not possible to set a correct ordering constraint between both actions. There are two different justifications for this situation:

- If z_1 and z_2 (or just one of them) were OPEN literals at some time during the second stage and no action deleted these literals during the process, the algorithm will have assumed the lowest level as their producer literal level. In some cases this is not the correct option and the consequence is that the subset of actions in the basic graph do not represent an optimal solution. This type of conflict is named *effect conflict* and it is usually due to a lack of information during the second stage. The algorithm will choose then a different producer level for z_1, z_2 or both at this stage. In order to solve an effect conflict the planner carries out the following operations:
  a) eliminate the action that achieves the literal in the current solution
  b) take the next upper level as the producer level for the literal
  c) resolve the process as usual by selecting an action in the new level
- If the only possible way to satisfy z_1 and z_2 is by means of actions a_1 and a_2 respectively, then we say this is a *precondition conflict*. The name comes from the fact that the literals involved in the conflict are the preconditions of the actions (in the example, a_1 needs literal y and deletes y and likewise for a_2). In this case, the literal in conflict has to be achieved again by a new action (from the problem graph) or an existing action (from the basic graph). Notice that this is the same operation the planner carries out when solving an action precondition. The only additional checking is to discover the correct ordering for the new action a_3 ($a_1 \rightarrow a_3 \rightarrow a_2$ or $a_2 \rightarrow a_3 \rightarrow a_1$).

The third stage is mainly devoted to solve ordering conflicts among the actions at the same level in the basic graph. In order to do this, the algorithm performs operations like introducing new actions to solve conflicts or replacing one action by another one for achieving a particular literal.

The partial consistency property allows for detecting unsolvable problems. If an effect conflict or precondition conflict cannot be resolved by any means then the problem is unsolvable. Notice that all of the actions, either in the basic graph or problem graph, are considered when solving a conflict. Therefore, an irresoluble conflict leads to an unsolvable problem.

The issue of optimality is related to these two points:

- As it was said above, the algorithm always applies criteria so as to generate the minimal set of actions: selecting firstly actions in the basic graph over those in the problem graph and preferring actions which have the less number of preconditions unresolved.
- An effect conflict implies there is an alternative solution for achieving a literal. The algorithm tends to use actions at the lowest levels for those OPEN literals which are never deleted by the propagation of effects. This behaviour may yield a non-optimal solution because a non-correct action level may be chosen for an action. The third stage is aimed at solving this problem.

4.1 Example: Monkey test 1

Let's apply property 1 to the basic graph obtained for the *hanoi* problem. It is possible to set a total-order relation between the two actions at $A3$ by ordering Move B P1 P3 before Move M P2 B (the latter deletes the precondition 10 of the former action). And for the two actions at $A2$, the consistent order is to put Move M B P2 before Move S P3 M. Consequently, the basic graph from the *hanoi* problem satisfies property 1 and no further operations are needed for this problem at this stage. Obviously, there are two missing actions in this partial solution but they will be discovered by the POP at the fourth stage.

A_1		L_1		A_2	
GoTo P1 P2	P={1,2} E={6}	M1 onfloor	1	GoTo P2 P1	P={1,6} E={2}
GoTo P1 P3	P={1,2} E={7}	M1 at P1	2	GoTo P2 P3	P={1,6} E={7}
GoTo P1 P4	P={1,2} E={8}	Box at P2	3	GoTo P2 P4	P={1,6} E={8}
		Ban at P3	4	GoTo P3 P1	P={1,7} E={2}
		Knf at P4	5	GoTo P3 P2	P={1,7} E={6}
		M1 at P2	6	GoTo P3 P4	P={1,7} E={8}
		M1 at P3	7	GoTo P4 P1	P={1,8} E={2}
		M1 at P4	8	GoTo P4 P2	P={1,8} E={6}
				GoTo P4 P3	P={1,8} E={7}
				Climb P2	P={3,6} E={9}
				PBox P2 P1	P={1,3,6} E={2,10}
				PBox P2 P3	P={1,3,6} E={7,11}
				PBox P2 P4	P={1,3,6} E={8,12}
				GetKnf P4	P={5,8} E={13}

Table 4. Partial problem graph for the *monkey* problem

Num	Literal	Num	Literal
1	M1 onfloor	9	M1 onbox P2
2	M1 at P1	10	B1 at P1
3	Box1 at P2	11	B1 at P3
4	Bananas at P3	12	B1 at P4
5	Knife at P	13	M1 hasknife
6	M1 at P2	14	M1 onbox P4
7	M1 at P3	15	M1 onbox P3
8	M1 at P4	16	M1 onbox P4
		17	**M1 hasbananas**

Table 5. Literals in the problem graph from the monkey test1 problem

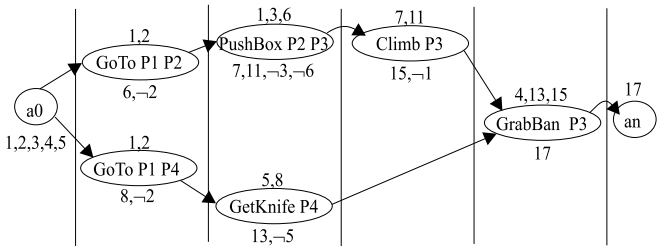


Figure 7. Basic graph for the monkey test 1 problem after stage 2.

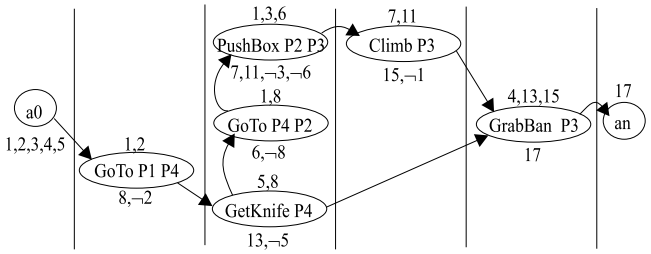


Figure 8. Final Basic graph for the monkey test 1 problem (after stage 3).

In order to illustrate the behaviour of the third stage (in particular, the effect conflict) we will take the *monkey and bananas* problem as an example. The literals in the problem graph are shown in Table 5, two action levels from the problem graph (levels $A1$ and $A2$) are shown in Table 4 and the basic graph is represented in Figure 7.

The first aspect to point out is that the set of actions in the basic graph does not yield an optimal solution. Actions Goto P1 P2 and Goto P1 P4 would force the introduction of an additional action like Goto P4 P1 or Goto P3 P1. When applying property 1 we notice there is a conflict at $A1$ as both movement actions require and delete literal 2.

During the process of creating the basic graph, literals 6 and 8 were OPEN literals as they are both generated by actions $A1$ and $A2$ (Table 4). Since no action at $A1$ would delete a precondition 6 or 8 of actions at $A2$ in the basic graph, the algorithm selected actions from $A1$ as the producer actions for literals 6 and 8 (the lowest level), and the consequence is that the comprised solution in the basic graph is non-optimal.

At the third stage, the algorithm proceeds to solve the effect conflict between the two actions at $A1$. The planner attempts to take $A2$ as the new producer level for literal 8 and applies the procedure for selecting an action. Two of the choices at $A2$ provoke again an ordering conflict (Goto P2 P4 and PushBox P2 P4 require and delete literal 6 and so conflict with action PushBox P2 P3 which also needs and removes literal 6); another choice would be Goto P3 P4 which does not satisfy the requirement of minimality because its precondition 7 is unresolved.

Subsequently, the planner checks what happens when attempting to find another way of solving literal 6 (leaving literal 8 at $A1$). There are two possibilities, one does not accomplish the requirement of minimality (Goto P3 P2) and the other choice does not involve any conflict (Goto P4 P2). Then the planner chooses Goto P4 P2 to replace the action producing literal 6 at $A1$. The final and optimal solution is shown in Figure 8.

5 THE FOURTH STAGE

As we explained above, the goal of the fourth stage is to obtain the final plan. In principle, the third stage in 4SP could be omitted and to execute directly the fourth stage after the second one. However, the task of finding the correct ordering constraints in the plan is accomplished in two steps (firstly ensuring a partial consistency in the graph and then finding an ordering for all of the actions in the graph) because important benefits can be gained:

- First and foremost to delay the use of a POP until it is strictly necessary. POP are very good at solving threats among actions, which is the final step of our planning algorithm (finding a final ordering relation among all actions in the plan).
- By finding a total-order relation between each pair of actions in the same action level it would be possible to obtain the final solution plan without having to execute the fourth stage.
- The third stage is also used as a way to verify the basic graph entails a solvable problem and an optimal solution. This point is very important since the POP would be unable to discover the solution is working with is non-optimal or there is no solution for the problem.

Our partial-order planner [8] is based on the UCPOP planner [7] and therefore completeness in guaranteed when starting from an empty initial plan. The POP is given the plan obtained at the third stage as an initial input plan. When the POP input is an empty plan, a complete search space is generated and all choices to solve an OPEN precondition or a conflict are considered in the resolution process. However, when the input is not an empty plan, completeness is not guaranteed because this non-empty plan is just the result of one branching line of the search space which would have been generated by a complete search method. A way to recover completeness in the POP is by means of the White Knight concept [3].

Hard interactions among actions in different sequences of different levels are not taken into account when building the basic graph. Therefore, it might be impossible to establish an ordering relation for the set of all actions in the basic graph. This means that, when a precondition p of an action a_j is deleted by one action $a_i, A_i < A_j$, which belongs to another sequence of actions, p must be restored by a new action (application of the white knight technique). This situation gives rise to two different types of basic graphs. Let $\mathcal{A}$ be the set of actions in a basic graph and $\mathcal{S}$ be the set of actions that constitute

a solution plan for a given problem:

- *complete basic graphs*, when $\mathcal{A} = \mathcal{S}$. In this case, a total order relation can be established among all actions in $\mathcal{A}$.
- *incomplete basic graphs*, when $\mathcal{A} \subset \mathcal{S}$. In this case, there are inconsistencies between actions of different sequences. New actions would have to be added to solve these interactions.

The difference between complete and incomplete basic graphs is specially important from the point of view of the fourth stage. In the case of complete basic graphs, the only remaining task is to sort the actions in the basic plan, whereas in the case of incomplete basic graphs, some actions will have to be added. In both cases, the remaining operations (ordering between actions and addition of new actions) are discovered by the existence of threats between the steps of the plan.

6 EXPERIMENTAL RESULTS

Due to a lack of time, the experiments shown in table 6 [5] correspond to a previous prototype of 4SP where the third stage is not implemented. Therefore, 4SP is not obtaining the optimal solution for problems such as monkey test 1, and it is not able to detect unsolvable problems.

Problems were taken from the UCPOP suite and Blackbox software distribution. All tests were run on a Sun Ultra 10 machine and results are given in seconds. We have run 4SP, BlackBox v3.6 [5] and STAN [4]. The results are classified into two groups: those for complete graphs and those for incomplete graphs (Table 6).

Problem	**Blackbox**	**STAN**	**Our method**	
Complete graphs	TT	TT	GT	TT
Sussman	0.02	0.028	0.005	0.006
Tw_rever4	0.03	0.03	0.011	0.021
Tw_rever5	0.06	0.03	0.023	0.04
Tower4	0.07	0.032	0.013	0.013
Tower5	0.21	0.07	0.024	0.025
Tower6	0.6	0.16	0.046	0.047
Tower9	111	10.53	0.225	0.225
T_largeA	0.82	0.53	0.079	0.08
T_largeB	4.34	2.63	0.289	0.3
T_largeC	—	82.143	1.792	1.8
T_largeD	—	—	4.508	4.52
Incomplete Graphs	TT	TT	GT	TT
Hanoi3d	0.11	0.039	0.019	0.176
Hanoi4d	1.41	0.061	0.035	1.81
Ferry	0.04	0.012	0.005	0.073
Monkeyt1	0.11	0.022	0.009	0.08
Monkeyt2	0.26	0.037	0.014	0.212

Table 6. Performance of Blackbox, STAN and our method on different problems

In most of the problems where 4SP was able to obtain a complete graph, the CPU time was reduced more than 50% compared to STAN and BlackBox. For example, in the blocks world domain, as the number of blocks increases, this difference is greater. This is specially remarkable in TowerLarge problems: 4SP was able to solve TowerLargeD problem that neither STAN nor BlackBox were able to.

For those problems with an incomplete graph, 4SP behaves slightly worse that STAN and BlackBox, although this difference is not as significant as in the case of complete graphs. As the average and standard desviation results show, 4SP behaviour is much more stable.

	Our method	**STAN**	**BlackBox**
Average	0.454	6.025	7.034
Standard Desviation	1.017	20.469	26.812

7 CONCLUSIONS AND FUTURE WORK

We have presented in this paper our four-stage planner 4SP. 4SP relies on the combination of an incremental preprocessing technique based on graph analysis and a POP. The basic graph is used to build a skeletal plan which is the POP's input. The most relevant aspect in 4SP is that the basic graph obtained with this graph-based technique already comprises the final solution plan for most of the tested domains.

Our objective was to develop a new planning approach by taking advantage of partial-order planning properties and reducing the inefficiency caused by the large search spaces generated by these planners. We have also shown that 4SP average outperforms other planning approaches as Graphplan or SATPLAN planners.

This is a first prototype of our planner 4SP. The obtained results confirm that the POP is still a bottleneck mainly for those problem which give rise to an incomplete graph. For this reason we suggest that the introduction of the third stage will significantly reduce the amount of work done by the fourth stage.

REFERENCES

[1] A. Barret, D. Christianson, M. Friedman, K. Golden, S. Penberthy, Y. Sun and D. Weld. *UCPOP v4.0 user's manual*, Technical Report TR 93-09-06d, Dept. of Computer Science and Engineering, University of Washington, Seattle, WA, (1996).

[2] A. Blum and M. Furst, *Fast planning through planning graph analysis*, Artificial Intelligence, **90(1-2)**, 281–300, (1997).

[3] D. Chapman, *Planning for Conjuntive goals*, Artificial Intelligence, **32-3**, 333–377, (1987).

[4] M. Fox and D. Long, *STAN and TIM public source code*, http://www.dur.ac.uk/CompSci/research/stanstuff/planpage.html, (1999).

[5] H. Kautz and B. Selman, *Blackbox Planner. Version 3.6*, http://http://www.research.att.com/ kautz/blackbox/, (1999).

[6] A. Howe, E. Dahlman, C. Hansen, M. Scheetz and A. von Mayrhauser, *Exploiting Competitive Planner Performance*, Proc. of the European Conference in Planning, 60–72, (1999).

[7] J.S. Penberthy and D.S. Weld, *UCPOP: A Sound, Complete, Partial Order Planner for ADL*, Proc. of the 1992 International Conference on Principles of Knowledge Representation and Reasoning, 103–114, (1992).

[8] L. Sebastia, E. Onaindia and E. Marzal, *Improving expressivity and efficiency in Partial-Order Causal Link Planners*, Proc. of the 18th Workshop of the UK Planning and Scheduling Special Interest Group, 124–136, (1999).

[5] GT stands for the time used in the graph creation and TT for the total time. We have used a blocksworld domain with 3 operators.

Towards Agent-Based Multi-Site Scheduling

Jürgen Sauer[1], Tammo Freese[2], Thorsten Teschke[3]

Abstract. Scheduling problems are usually treated within single plant environments or within companies with several production locations. Due to the globalization of markets companies can no longer be regarded isolated from each other. They are in fact elements of spatially distributed production and logistics networks, where apart from the actual production process transport and stock keeping gain importance. This contribution analyses the organizational structures found in distributed production networks and proposes an approach for their mapping to multiagent systems for integrated production planning and scheduling. Moreover, a platform for multiagent systems deployment is outlined, which is apt to satisfy the major requirements to an agent platform for dynamic, distributed production and logistics networks.

1 INTRODUCTION

In the past companies have often been regarded as self-contained units with well-defined business relationships to consumers and suppliers. Measures of optimization were confined by a company's boundaries. Today, competencies for fast and economical development and manufacturing of complex products are distributed to different companies. This trend as well as the markets' evolution to buyer's markets and shortened product life cycles necessitate optimizations beyond a company's boundaries. Concepts like *supply management*, *supply chain management* and eventually *virtual enterprises* have been devised in order to open up new potentials of optimization by regarding a company's interweavement with its suppliers and consumers ([1; 2]).

2 MOTIVATION AND PROJECT OUTLINE

The globalization of markets leads to the formation of spatially distributed production and logistics networks. In addition to the actual production process transport and stock keeping are of increased importance, since they strongly affect a company's ability to meet delivery deadlines. For this reason centralized approaches to production planning and scheduling for companies with a single production site cannot be transferred to distributed enterprises directly. Moreover, schedule stability decreases in centralized approaches, since even locally resolvable perturbances like machine breakdowns change the central schedule. *Multi-site scheduling* ([3]), instead, represents a promising approach where planning and scheduling encompasses two levels of hierarchy. On the upper, global level a rough schedule for the entire enterprise is generated based on imprecise, cumulated information on available resource capacities. This rough schedule defines targets for the enterprise's individual sites, which refine it into site-specific schedules. The distribution of the planning and scheduling process effects an increased schedule stability, since locally resolvable deviations do not have to be regarded within global planning and scheduling.

The AMPA project (Agent-Based Multi-Site Planning and Scheduling Application Framework) is concerned with distributed planning and scheduling in dynamically changing logistics networks ([4]). Starting from the notion of multi-site scheduling, the dynamics of business relationships and the individuality of companies within a logistics network has to be considered by pursuing a multiagent approach. According to [5] multiagent systems represent a suitable abstraction for modelling scheduling problems. In order achieve the goals stated above, the multi-site scheduling approach will be enhanced in many respects. An integrated consideration of production and transport planning and scheduling is striven for. Moreover, the restriction to two levels of planning and scheduling hierarchy abolished, yielding a more detailed decomposition of the planning and scheduling problem. Finally, the strict hierarchical decomposition of planning and scheduling problems regarded in multi-site scheduling is complemented by a network dimension which is especially suitable for representing inter-company relationships. This imposes negotiation tasks in vertical (along the hierarchy) as well as in horizontal (between departments of different firms) direction.

Agents to be employed in the production, transport and stock keeping domain differ regarding the knowledge as well the heuristics and strategies used for performing their planning, scheduling and coordination tasks. The development of software agents and a respective agent platform within the AMPA project will therefore pursue a component-based approach ([6]). Specific types of agents can then be created by exchanging and configuring software components (cf. [7] and [8]).

In order to realize such an approach three major steps have to be performed. First, an adequate model of the organizational structures with respect to multi agents has to be found. Second, the planning and scheduling requirements and capabilities of the agents on the different levels of the organizational model have to be defined. And third, a communication model between the agents along the hierarchy as well as between agents of different company substructures has to be developed. The following chapters focus on the first step and describe the basic notions of the problem area as well as the organizational model for the design of an agent-based multi-site scheduling system.

[1,2,3] Fachbereich Informatik, Universität Oldenburg, Escherweg 2, D-26121 Oldenburg, Germany, email: sauer@informatik.uni-oldenburg.de, http://www-is.informatik.uni-oldenburg.de/~sauer

3 BASICS

Within the scope of the preceding outline of the AMPA project's contents the major subjects "supply chain management", "virtual enterprises" and "software agents" and "multi-site scheduling" have emerged. These shall be illustrated more detailed in the subsequent sections.

3.1 Multi-Site Scheduling

Scheduling problems are usually treated in a single plant environment where a set of orders for products has to be scheduled on a set of machines [9-12]. However, within many industrial enterprises the production processes are distributed over several manufacturing sites, which are responsible for the production of various parts of a set of final products. Usually, there is no immediate feedback from the local plants to the logistics department and communication between the local schedulers takes place without any computer-based support.

Due to the distribution of production processes to different plants some specific problems arise in addition to the problems of the dynamic complex scheduling environment:

- Interdependencies between production processes that are performed in different plants have to be regarded.
- In global scheduling generalized and inprecise data are used instead of precise data.
- Existing (local) scheduling systems for individual plants that accomplish the local realization of global requirements should be integrated.
- The coordination of decentralized scheduling activities for all plants within one enterprise is necessary since several levels of scheduling with their specific scheduling systems have to work cooperatively in a dynamic distributed manufacturing environment.
- The uncertainty about the actual "situation" in individual plants has to be regarded.
- Different goals have to be regarded on the different levels.

The multi-site scheduling approach [3] presents a hierarchical two-level structure reflecting the organizational structure often found in business.

On the upper global level requirements are generated for intermediate products manufactured in individual locations. Local scheduling (at individual locations) deals with the transformation into concrete production schedules which represent the assignment of operations to machines. On both levels predictive, reactive as well as interactive problems are addressed, not only to generate schedules but also to adapt them to the actual situation in the production process. Additionally, communication between the systems is needed to support the consistent exchange of data and to coordinate the local scheduling systems.

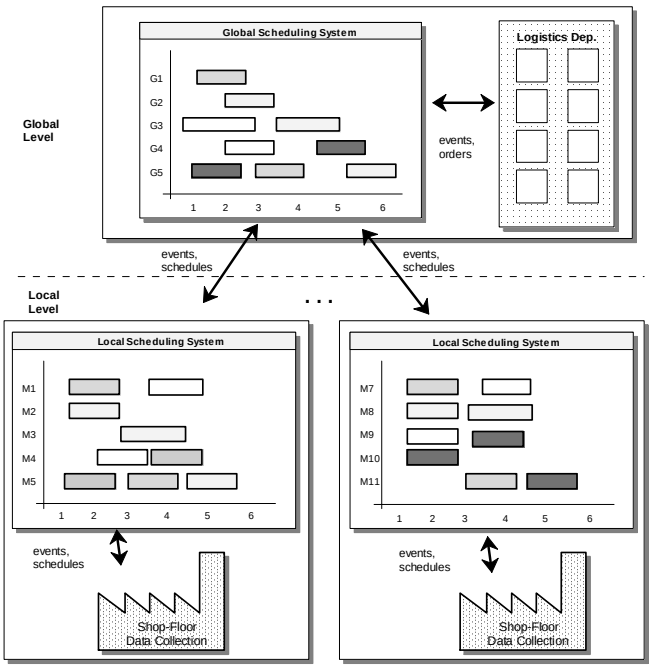


Figure 1. Multi-Site Scheduling

The multi-site scheduling tasks can be characterized as follows:

- *Global predictive scheduling*: A global-level schedule with an initial distribution of internal orders to local production sites is generated.
- *Global reactive scheduling:* If problems cannot be solved on the local level or the modified local schedule influences other local schedules (inter-plant dependencies), global reactive scheduling can then cause a redistribution of internal orders to local plants and adapt the global schedule.
- *Local predictive scheduling:* Based on the global schedule, the local plants draw up their detailed local production schedules.
- *Local reactive scheduling:* In case of local disturbances, the local reactive scheduler first tries to remedy them locally by interactive repair.
- *Communication and coordination:* Both levels have to be provided with data as actual and consistent as possible. Therefore information has to be sent between the lecels, e.g. the global schedule consisting of information on internal orders, affiliated intermediate products, machine groups to use, time windows that should (possibly) be met, and required quantities of intermediate products, unexpected events that effect the local resp. the global level (e.g. the cancellation of an order or breakdowns of machine groups).

For the solution of the predictive and reactive scheduling tasks several problem solving approaches are useful. Some of them have been checked for the MUST (Multi-Site Scheduling System) approach [3]. Table 1 shows the tasks and some of the appropriate methods from which several are investigated in the MUST project.

Table 1: Multi-Site Scheduling Tasks and Methods

Problem Area	Techniques
Global Predictive Scheduling	Heuristics, Constraints, Genetic Algorithms, Fuzzy-Logic
Global Reactive Scheduling	Interaction, Heuristics, Constraints
Local Predictive Scheduling	Constraints, Heuristics, Genetic Algorithms, Neural Networks, OR-Systems
Local Reactive Scheduling	Interaction, Heuristics, Constraints, Multi-Agents
Communication	Blackboard, Contract Net

The approach is implemented in the distributed knowledge-based scheduling system MUST. The system architecture reflects the two level approach and consists of one global scheduling subsystem and several local subsystems, one for each individual production site. All systems include the knowledge-based techniques described for the predictive and reactive scheduling tasks to be performed. Communication is realized using the blackboard paradigm. The MUST subsystems are implemented as decision support systems thus lacking one of the major characteristics of multi-agent systems, which is the proactivity (see 3.4).

3.2 Supply Chain Management

Decreasing transaction costs, advanced control of processes and thinking in profit centers increasingly lead to companies outsourcing those parts of the creation of value without core competencies. The growing force to shorten delivery periods and product innovation cycles while at the same time increasing the rates of return induced by globalized markets requires an intensified cooperation of all companies along the inter-company supply chain ([1]).

The concept of a supply chain is insofar misleading as the companies involved in the development and manufacturing as well as transport, distribution and selling of a product usually do not constitute a chain, but rather a network. Supply chain management coordinates the activities within this logistics network under the overall goal of inter-company and intersite optimization of a product's development and manufacturing process as well as the innovation of processes.

Characteristic for supply chain management is the strategic, long-term cooperation of companies as well as the small number of suppliers for a particular product. Cooperations according to the supply chain management approach rely on massive exchange of information, which presupposes trust between the partners within the supply chain and the long-term abolishment of information barriers between the individual companies.

Among the risks of supply chain management is the development of unilateral dependencies and the potential abuse of information on co-producers. Additionally, due to its long-term orientation the concept of supply chain management is not suitable for short-term cooperations ([2]).

3.3 Virtual Enterprises

A virtual enterprise represents a network of companies affiliated in order to perform a particular, temporally limited task, thereby appearing as an entity. Opposed to the concept of supply chain management, which regards strategic relationships of rather long-term nature, virtual enterprises excel especially by their ability to flexibly reorganize themselves. A virtual enterprise can be viewed as a temporarily existing supply chain.

Virtual enterprises are established in order to be able to flexibly react to the opportunities of highly dynamic markets. The selection of partners is based on cost effectiveness and uniqueness of their products instead of more traditional factors like organizational size, geographic location, IT infrastructure, employed technologies and implemented processes. The companies engaged in a virtual enterprise share their knowledge, their competencies and business relationships in order to perform the virtual enterprise's task. This combination of forces is to enable the companies to reach global markets with products and solutions that each of them could not have accomplished on its own ([2]).

3.4 Software Agents

Software agents represent a software development paradigm which is appropriate for distributed problem solving. They are employed in numerous systems of distributed artificial intelligence (DAI). In common linguistic usage, an agent is everyone who acts on behalf of another. In computer science the concept of a software agent is not uniformly defined. In [13] Franklin and Graesser present a comparison of numerous definitions.

Wooldridge defines an agent as "a computer system that is situated in some environment, and that is capable of autonomous action in this environment in order to meet its design objectives" ([14]). Therefore, a fundamental property of an agent is *autonomy*: an agent operates without direct interference by humans or other systems, and has control over its behaviour and its internal state. The concept of an intelligent agent extends this definition by the capability of acting flexibly, whereby the notion of flexibility comprises three characteristics:

- *reactivity*: agents perceive their environment and react timely and appropriately to changes within this environment;
- *pro-activeness*: agents do not only react to observed changes within their environment, but are capable of taking the initiative in a goal-directed fashion;
- *social ability*: agents interact with other agents (and possibly humans) by exchanging information formulated in a mutually agreed communication language. Moreover, the notion of social abilities comprises complex patterns of behaviour based on communication protocols, e.g. for the purpose of negotiation.

This concept of intelligent agents is perfectly suitable for the domain of production planning and scheduling. First, an agent has some kind of knowledge of the problem to be solved (the scheduling problem) and its environment (e.g., other agents or the shop-floor), and is capable of negotiation. Second, it is able to quickly react to changes within its environment, e.g. a machine breakdown. And third, agents are pro-active, allowing them, e.g., to improve their schedules while no other service request are issued [15]. Therefore, this definition is adopted for the agents to be developed within AMPA: Every agent shall be able to schedule its activities (autonomy), to change its schedule in case of disturbances (reactivity), and to optimize its schedule (pro-

activeness). Messages concerning changes, disturbances etc. are exchanged using a communication language commonly agreed upon (social abilities).

Additional features of agents that are studied in different approaches of DAI are ([16]):

- *mobility*: mobile agents are able to move within electronic networks;
- *veracity*: a truthful agent does not knowingly provide other agents with false information, e.g., on its environment or its internal state;
- *benevolence*: benevolent agents do not have conflicting goals, and they try to achieve what they are asked to;
- *rationality*: rational agents try to achieve their goals. They do not knowingly act in a way conflicting with their goals.

The central problem of multiagent systems is how to achieve coordinated action among agents in a way yielding problem solving capabilities that exceed those of any individual agent.

4 MAPPING ORGANIZATIONAL STRUCTURES TO SOFTWARE AGENTS

The basic elements of business organizations are *posts* and *relationships* between them. Important dimensions of business organization systems are specialization, coordination and the directional system. The aspect of specialization is concerned with the division of work, which is about different organizational units performing partial tasks of different kinds. The division of work attained by specialization requires the coordination of its entailed activities. This task can be simplified by means of hierarchies. The directional system specifies authorities to instruct, responsibilities, and powers of decision which a superordinated post has regarding to a subordinated post. Concerning the structure of posts within the directional system two typical basic forms can be distinguished. The single-line system rests on the principle of unity of command and organizes posts in a tree structure. The multiple-line system aims at realizing the principle of shortest paths in interdepartmental coordination problems and organizes posts in a graph structure. In the latter form, a post can be subordinated to several posts within the hierarchy. This, however, may entail questions of authority and the risk of unclear responsibilities ([17]).

4.1 Organization Model

The organization model proposed by AMPA enhances the multi-site scheduling approach delineated in section 3.1 in two respects: first, the hierarchy considered by multi-site scheduling is extended by additional levels, and second, the hierarchical, intra-company perspective is complemented by a network-like, inter-company dimension.

Hierarchical structures are a common representation for intra-company directional systems for they are suitable for defining powers of decision, authorities to instruct, duties of supervision and tasks of inspection. This approach is also pursued within the scope of AMPA. Therefore, posts are defined according to resource-oriented aspects and arranged in a hierarchy. Potential posts are, e.g., an entire company, production sites, job shops, warehouses, transport vehicles, resource groups, or machines. A post is represented by a planning agent.

Complex organization structures, however, are not exclusively organised hierarchically. This applies especially to legally and economically independent enterprises, for relationships between them do neither define powers of decision nor authorities to instruct; they only represent the aspect of coordination between their directional systems. In addition to the hierarchical, static dimension of the intra-company directional system in AMPA the network-like, dynamic dimension of the coordinating, logistical relationships on all levels of hierarchy is considered by a special type of relationship. Within AMPA organizations are accordingly represented by an overlay of hierarchical and network-like structures, thus achieving both vertical and horizontal integration. The resulting organization model is illustrated by Figure 2, where the enterprise under consideration is represented by dark nodes. These nodes also represent the problem area of multi-site scheduling. External organizational units are depicted using pale nodes.

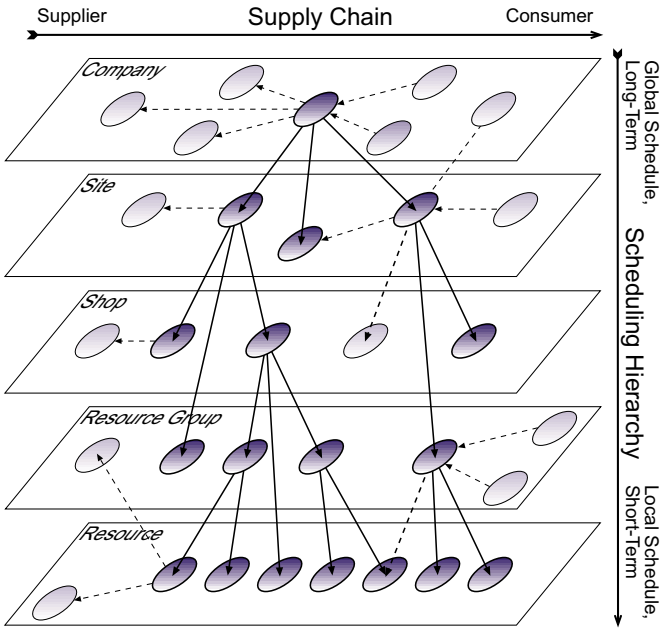


Figure 2. Organization Model

On a more formal level the approach pursued within AMPA can be regarded as a combination of the single-line system and the multiple-line system. This combination aims at clear responsibilities on the one hand and improved ways of coordination by shorter communication paths on the other. In order to achieve these goals while avoiding the disadvantages pointed out before two different types of relationships are considered. Both are directed and define a potential usage of a post by another, whereby subsequently the former post will be referred to as *supplier* and the latter as *consumer*. Usage as indicated by such a relationship is thus regarded as the supplier providing services for the consumer.

Disciplinary subordination relationships are part of the directional system and serve the representation of the organization model's hierarchical intra-company dimension. In addition to potential usage of a post by another they specify the sole responsibility a superordinated post (the consumer) has for its disciplinarily subordinated posts (the suppliers). In order to achieve the therefore required unambiguousness a post may

maximally be subordinated to one other post, i.e. the structure of the disciplinary subordination relationships correspond to the single-line system. The responsibility of a superordinated post for its disciplinarily subordinated posts is expressed by the latter posts' opportunity to report order requests to the former post, if these requests cannot be accomplished by a service provided by a post subordinated to the latter. Moreover, a disciplinary subordination relationship requires a subordinated post to grant access to information on its state (e.g., its workload) to its superordinated post. This information may affect the superordinated post's decision making process. In Figure 2 disciplinary relationships are depicted by solid arrows.

Functional relationships represent the network-like dimension of the organization model and serve the coordination between organizational units which are not connected by the directional system. A functional relationship solely defines a usage relationship between two posts, i.e. the supplier may neither forward requirements to the consumer, nor does the consumer have any kind of responsibilities for the supplier. Moreover, the supplier is not bound to give away information on its state to the consumer. In addition to one disciplinary subordination relationships a post may engage in arbitrary functional relationships. Hence, the system of functional relationships corresponds to the multiple-line system. In Figure 2 functional relationships are represented by broken arrows.

4.2 Mapping to Software Agents

In the AMPA project an enterprise is represented by a system of agents. This allows a direct mapping of the enterprises' structures and their communication as well as the integration of existing scheduling applications. The following steps describe how an enterprise is mapped onto a multiagent system.

1) *Identify agents:* Every entity in an enterprise for which AMPA should do the scheduling is represented by an agent. However, if a scheduling system for one or more of the entities exists, it need not be replaced, but it is represented by a single agent which acts as a wrapper. Therefore existing scheduling systems like the subsystems of MUST may be integrated easily which leads to an open scheduling environment.
2) *Define scheduling tasks*: Depending on the position in the network different planning and scheduling tasks have to be performed by the agents. These tasks have to be identified and the appropriate scheduling knowledge, e.g. one of the algorithmic solutions presented in table 1, has to be added to the agent.
3) *Add disciplinary relations:* Between the Units which are represented by the agents identified in step 1, there are disciplinary relations. These organize the agents in a tree structure. If the network of disciplinary relationships between the agents has no tree structure, exactly one of the relationships must be chosen for having a unity of command.
4) *Add functional relationships:* Aside disciplinary relations, there are also functional relations between the units in an enterprise. These should be adopted in the agent system.

The first three steps yield an agent structure that represents the internal structures of an enterprise. The following step is to embed this enterprise into its environment.

5) *Integrate suppliers and consumers:* To achieve an integration of an enterprise into its logistics network, the suppliers and consumers of the enterprise have to be represented in the agent system. If they also use AMPA agents, the correct agents must be identified. Otherwise, wrapper agents encapsulate the communication with these business associates. If a large group of similar suppliers or consumers has to be integrated, it can alternatively be represented by a single agent which wraps the communication with all the members of the group. The agents are integrated in the agent structure by adding functional relationships to the agents for which they are suppliers or consumers. For both communication possibilities an appropriate communication model on the basis of the contract net protocol has to be defined.

4.3 A platform supporting the agents

For deployment of the agent system, an agent platform is needed that acts as an environment for the agents. It aims in supporting the flexible and easy configuration of new agents to be incorporated into the whole system. Based on a Java virtual machine the platform supports amongst others:

- *Distribution (agent context):* The agents of an AMPA are normally distributed in two different kinds. A group of agents which runs on the same server is *locally distributed*. In a *global distribution,* the agents run on different servers. As an example, the agents of an enterprise run on one server, whereas the enterprises which build a virtual enterprise or a supply chain will normally run servers on their own.
- *Communication (communication layer):* To support different kinds of distribution, there must be different kinds of communication between local and global distributed agents. An agent platform should have a communication interface that encapsulates that hides this difference from the agents. So an agent has no information whether its communication partners reside on the same or on a different server.
- *Platform independence:* For deploying an agent system in an heterogenous environment of computer systems, it is either required to develop specific agents for every platform, or to use a platform that offers an equal interface on all systems. The first possibility is not suitable especially in virtual Enterprises, where a huge number of different systems is to expect. Therefore it is more convenient to develop an agent platform in Java which would be independent of the underlying hardware structure.
- *Security (security policy):* An agent represents a real system and acts at least in parts autonomously. In this context, security must have a high priority. An agent platform has to implement suitable security mechanisms that protect the agents from unauthorized access. In contrast, authorized users must have full control over their agents. To achieve a combination of these requirements, the platform must have an authorization concept that offers roles with different access grants.
- *Persistency:* An agents needs access to persistent information like the production schedule or the product ontology. To achieve this in combination with platform independence, the agent platform has to decouple the application layer (the agent) and the persistency layer.

- *User interfaces:* In most cases, the possibility to observe and agent and to intervene in its actions is very important for the acceptance of an agent system. Therefore an agent system should support connections of agents and user interfaces.
- *Transactions:* Complex scheduling processes involve many subsequent negotiations with agents along the supply chain. Sometimes they can only be executed partly what leads to an inconsistent schedule. The agent platform should provide a transaction concept which allows the rollback of failed transactions.
- *Configuration (configuration information, resources):* The persistency and security mechanisms require a possibility to replace underlying database management systems and ERP systems. Moreover, configuring the deployed agents is necessary.

Figure 3 shows an architecture outline that targets to fulfil the requirements above.

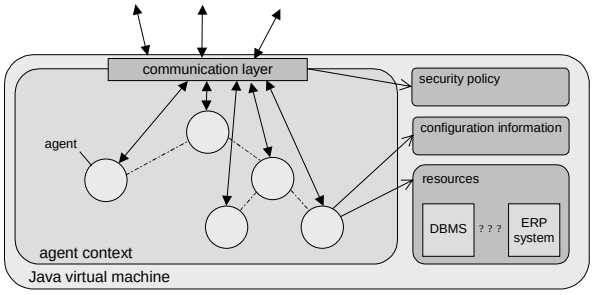


Figure 3. Architecture of agent platform

The component structure of the platform and the agents used within the platform allow for the easy configuration of different systems. On the basis of a generic agent it is possible to generate specialized agents for production, transport or stock. This is realized by the exchange and configuration of components of the agents [7].

4.4 Related work

A comparable approach is pursued by Swaminathan et al. in [7], who represent the structural elements of a supply chain like, e.g., production and transport units by agents. In contrast to the organization model proposed in this section hierarchical relationships are not considered, i.e. a supply chain is modelled as a flat network.

Other agent-based approaches for enterprise modeling often focus on the definition of ontologies supporting the description of the workflows within the enterprises [18]. Within the "Enterprise"-project agents are used to represent tools that perform activities. These agents are integrated in a system for workflow management. The TOVE project [19] uses agents to represent the "classical" functions of production planning and -control. In addition these agents are connected via information agents providing the necessary information for the "functional" agents involved.

5 CONCLUSION AND FURTHER WORK

On the basis of the multi-site scheduling approach an extension in several directions is proposed. Not only one enterprise with distributed production has to be considered but also the suppliers of the several units shall be integrated in the scheduling task. The systems supporting the scheduling tasks are organized as agent-based systems thus offering all the advantages of multi-agent systems. The presented approach for an agent-based system performing scheduling in production networks is still under development. First steps have been the modeling of the organizational structure for a multi-agent system. The next steps are the prototypical implementation of the agent platform and a number of agents using a common framework like Enterprise Java Beans [20] or tools for the development of multi-agent systems like ZEUS [21]. Within this prototype the position depending scheduling knowledge and the extended negotiation protocols will be integrated and tested.

6 REFERENCES

[1] Scheer, A.-W., and Borowsky, R. 'Supply Chain Management: Die Antwort auf neue Logistikanforderungen'. *LM'99 - Intelligente I+K Technologien*, Bremen. Springer Verlag. 1999, pp. 3-14.

[2] Schönsleben, P. *Integrales Logistikmanagement: Planung und Steuerung von umfassenden Geschäftsprozessen*. Springer, Berlin, Heidelberg. 1998.

[3] Sauer, J. 'A Multi-Site Scheduling System'. *AAAI's Special Interest Group in Manufacturing Workshop on Artificial Intelligence and Manufacturing: State of the Art and State of Practice*. 1998.

[4] Appelrath, H.-J., Freese, T., Sauer, J., and Teschke, T. 'Konzept eines Multiagentensystems für die verteilte Ablaufplanung'. *Proceedings des Workshops "Agententechnologie" auf der KI '99*, Bonn. 1999, pp. S. 37-45.

[5] Henseler, H. *Aktive Ablaufplanung mit Multi-Agenten*. Dissertation, Carl von Ossietzky Universität Oldenburg, Oldenburg. 1998.

[6] Szyperski, C. *Component Software - Beyond Object-Oriented Programming*. Addison-Wesley Verlag. 1998.

[7] Swaminathan, J. M., Smith, S. F., and Sadeh, N. M. 'Modeling Supply Chain Dynamics: A Multiagent Approach'. *Decision Sciences Journal*, 29(3). 1998, pp. 607-632.

[8] Sauer, J., Appelrath, H.-J., Bruns, R., and Henseler, H. 'Design-Unterstützung für Ablaufplanungssysteme'. *KI-Künstliche Intelligenz*, 2/97. 1997, pp. 37-42.

[9] Dorn, J., and Froeschl, K. A. *Scheduling of Production Processes*. Ellis Horwood. 1993.

[10] Sauer, J., and Bruns, R. 'Knowledge-Based Scheduling Systems in Industry and Medicine'. *IEEE-Expert*(February). 1997.

[11] Smith, S. F. 'Knowledge-based production management: approaches, results and prospects'. *Production Planning & Control*, 3(4). 1992.

[12] Zweben, M., and Fox, M. S. *Intelligent Scheduling*. Morgan Kaufman. 1994.

[13] Franklin, S., and Graesser, A. 'Is it an Agent, or just a Program?: A Taxonomy for Autonomous Agents'. *Third International Workshop on Agent Theories, Architectures, and Languages*. Springer-Verlag. 1996.

[14] Wooldridge, M. 'Intelligent Agents'. *In:* G. Weiss, ed. *Multiagent Systems: A Modern Approach to Distributed Artificial Intelligence.* The MIT Press. 1999, pp. 27-77.

[15] Henseler, H. 'From Reactive to Active Scheduling by using Multi-Agents'. *In:* R. M. Kerr and E. Szelke, eds., *IFIP TC5/WG.5.7 Second International Workshop on Knowledge Based Reactive Scheduling (KBRS) PREPRINTS.*, Budapest. 1994.

[16] Wooldridge, M., and Jennings, N. R. 'Intelligent Agents: Theory and Practice'. *Knowledge Engineering Review*, 10(2). 1995.

[17] Schierenbeck, H. *Grundzüge der Betriebswirtschaftslehre*. Oldenbourg Verlag GmbH, München. 1993.

[18] Fox, M. S., and Gruninger, M. 'Enterprise Modeling'. *AI Magazine*, 19(3 (Fall 98)). 1998, pp. 109-121.

[19] Barbuceanu, M., and Fox, M. S. 'The Information Agent: An Infrastructure Agent Supporting Collaborative Enterprise Architectures'. *3rd Workshop on Enabling Technologies: Infrastructure for Collaborative Enterprises*, Morgantown, WV. 1994.

[20] Matena, V., and Hapner, M. *Enterprise JavaBeans Specification*., Sun Microsystems. 1999.

[21] http://www.labs.bt.com/projects/agents/zeus/

OCLGraph : Exploiting Object Structure in a Plan Graph Algorithm

R. M. Simpson, T. L. McCluskey and **D. Liu** [1]

Abstract. In this paper we discuss and describe preliminary results of integrating two strands of planning research - that of using plan graphs to speed up planning, and that of using object representations to better represent planning domain models. To this end we have designed and implemented OCL-graph, a plan generator which builds and searches an object-centred plan graph, extended to deal with conditional effects.

1 Introduction

This paper describes work that is part of a continuing effort to evaluate the impact of modelling planning domains in an object-centred way, using a family of planning-oriented domain modelling languages known as OCL [10]. The benefit is seen as twofold: (a) to improve the planning knowledge acquisition and validation process (b) to improve and clarify the plan generation process in planning systems. With regard to (b), it is our belief that certain obstacles and problems that researchers into planning algorithms encounter can be alleviated using a rich, planning-oriented knowledge representation language.

The object-centred language OCL, and more recently the hierarchical version OCL_h [8, 9], have their roots in the 'sort abstraction' ideas used in the domain pre-processing work of [12]. OCL is primarily aimed as a high level language for planning domain modelling, the main feature distinguishing it from STRIPS-languages being that models are structured in terms of objects, rather than literals. It aims to allow modellers to more easily capture and reason about planner domain encodings independent of planning architecture, and to help in the validation and maintenance of domain models. On the other hand, OCL retains all the flexibility of a STRIPS-like encoding. The rationale behind OCL has been sustained by the experience of those applying planning technology. For example, the developers of the planner aboard Deep Space 1 [11] stress the need to develop clean, planner-independent languages that can be used to build and statically validate domain models.

In this paper we seek to tie up the advantages in creating a domain model in OCL with the use of a particularly successful form of plan generation using a plan graph algorithm called Graphplan [2]. The plan graph has been used as the basis for many experimental planning systems, and was the the basis of most of the planners in the AIPS-98 planning competition. This paper describes our investigation into the use of an *object-centred plan graph* in a Graphplan-like planning algorithm. Parallel work [9, 6] is investigating the use of OCL in traditional plan-space search algorithms. The current effort is therefore part of a larger project to implement many of the best regarded planning algorithms in a manner both to process planning problems expressed in OCL and to develop the algorithms in a manner to take advantage where possible of the additional information content of OCL models.

After introducing the reader to OCL and Graphplan, we detail the design of a planner which draws from Graphplan in algorithmic details, and from OCL for its representation. We argue that the 'object-graph' algorithm embedded in OCL-graph is conceptually simpler than the corresponding literal-based algorithm. Also we have extended the algorithm to deal with conditional effects using a strategy similar to that used by [7] and to the factored expansion strategy described by [1].

Our results suggest that the use of OCL (i) simplifies the plan graph: proposition levels become object levels where it is implicit that an object can only be in one 'substate' at one time (ii) simplifies the detection of 'mutex' relations and (iii) provides a surprisingly natural way of dealing with conditional effects. Finally, our initial implementations using tests from standard toy benchmark domains suggest that there may be costs as well as benefits involved in using a rich domain model with existing planning technology.

2 Foundations of OCL

2.1 Overview

In OCL the world is populated with objects each of which exists in one of a well defined set of states (called 'substates'), where these substates are characterised by predicates. On this view an operator may, if the objects in the problem domain are in some minimal set of substates, bring about changes to the objects in the problem domain. The application of an operator will result in some of the objects in the domain moving from one substate to another. In addition to describing the operators in the problem domain OCL provides information on the objects, their object class hierarchy and the permissible states that the objects may be in. The main advantage of the OCL conception of planning problems to algorithms is that they do not need to treat propositions as fully independent entities rather they now belong to collections that can be manipulated as a whole. So instead of dealing with propositions the algorithms deal with objects (typically fewer objects than propositions). This is a type of abstraction which we believe most naturally co-insides with domain structure. It provides opportunities to improve on existing planning algorithms by adapting them to operate at the object level rather than the propositional level.

[1] Department of Computing Science University of Huddersfield, UK r.m.simpson@hud.ac.uk t.l.mccluskey@hud.ac.uk d.liu@hud.ac.uk

2.2 Basic Formulation

A domain modeller using OCL aims to construct a model of the domain in terms of objects, a sort hierarchy, predicate definitions, substate class definitions, invariants, and operators. Predicates and objects are classed as dynamic or static as appropriate - dynamic predicates are those which may have a changing truth value throughout the course of plan execution, and dynamic objects (grouped into dynamic sorts) are each associated with a changeable state. Each object belongs to a unique *primitive sort* s, where members of s all behave the same under operator application. In what follows we will explain those parts of OCL sufficient for the rest of the paper, the interested reader is referred to the bibliography for more information.

An 'object description' in a planning world is specified by a triple (s, i, ss), where i is the object's identifier, s is the objects *primitive sort* and ss is its $substate$ - a set of ground dynamic predicates which all $refer$ to i. All predicates in ss are asserted to be true under a locally closed world assumption.

As a running example we will use a version of the Briefcase World, as this is simple and has been used in [1] as the basis of their discussion on the implementation of conditional effects in 'Graph Plan'. Note that, however, this does not illustrate the full benefits of an OCL encoding as the briefcase world is structurally simple. Dynamic objects in a briefcase world could be of sort bag (identifiers briefcase,suitcase,..) or of sort thing (identifiers cheque,dictionary,suit,..), and static objects may be of sort location (identifiers home,office ..). Two examples of objects description are

```
(thing,cheque,[at_thing(cheque,home),
         inside(cheque,briefcase),
         fits_in(cheque,briefcase)])
(bag,briefcase,[at_bag(briefcase,home)])
```

A **world state** is a complete set of object descriptions for all the dynamic objects in the planning application, and is usefully viewed as a total mapping between object identifiers and their corresponding substates, as an identifier is allowed to be associated with exactly one substate. States are constrained by **invariants**. These define the truth value of static predicates and the relationships between dynamic predicates. In particular they are used to record inconsistency constraints. A world state that satisfies the invariants is called well-formed.

For each sort s, the domain modeller groups a sort's substates together, specifying each group with a set of predicates called a **substate class definition**. They form a complete, disjoint covering of the space of substates for objects of s. When fully ground, a substate class definition forms a legal substate. To ensure that any legal ground instantiation of a substate class definition gives a legal substate, definitions usually contain static predicates. The substate class definitions for the dynamic sorts $thing$ and bag in the briefcase world are:

```
substate_classes(thing,
   [at_thing(Thing,Location),
    inside(Thing,Bag),fits_in(Thing,Bag)],
   [at_thing(Thing,Location),outside(Thing)])
substate_classes(bag,
   [at_bag(Bag,Location)])
```

meaning that a thing can only be either at a location and in a bag that it fits into or that it is at a location but is not in any bag, and a bag must be positioned at a location. If i is a variable or an object identifier of sort s, and se is a set of predicates, then (s, i, se) is called an **object expression** if there is a legal substitution t such that $i_t = j$ and $se_t \subseteq ss$, for at least one object description (s, j, ss). The third component of an object expression is thus called a substate expression. Also, we define an **object class expression** (s, i, ce) to be an object description that may contain variables and static predicates in ce. When ground therefore, an object class expression becomes a valid object description if the static predicates it may contain are true in the domain model.

A **planning task** is defined by a well-formed world state, and a goal consisting of any legal mapping of object identifiers to substate expressions i.e. a goal is a set of object expressions with distinct objects identifiers.

2.3 Operator Representation

An **object transition** is an expression of the form $(s, i, se \Rightarrow ce)$ where i is a dynamic object identifier or a variable of sort s, and se and ce are such that (s, i, se) is an object expression and (s, i, ce) is an object class expression. If cc is an object transition, then we use the notation $cc.lhs$ and $cc.rhs$ to refer to se and ce respectively.

An action in a domain is represented by operator schema O with the following components: $O.id$, an operator's identifier; $O.prev$, the prevail condition consisting of a set of object expressions; $O.nec$, the set of necessary object transitions; and $O.cond$, the set of (conditional) object transitions. Each expression in $O.prev$ must be true before execution of O, and will remain true throughout operator execution. In the briefcase world we have operators put_in, take_out and move. The put_in operator will have a prevail section which allows us to specify that the bag is at a location L but this does not change as a result of applying the operator. The necessary section specifies that the thing must be at the same location as the bag and must be outside all containers prior to the application of the operator but as a result of applying the operator the thing will now be inside the bag but still at the same location. The operator can be specified as follows:

```
operator(put_in(T,B),
     % prevail
     [(bag,B,[at_bag(B,L)])
     ],
     % necessary
     [(thing,T,[at_thing(T,L),outside(T)]
                =>
              [at_thing(T,L),inside(T,B),
               fits_in(T,B)])
     ],
     % conditional
     [ ])
```

We define O.Pre to be the preconditions of O, i.e. the set of object expressions in $O.prev$ and the set of left hand sides of $O.nec$. Hence put_in.Pre is [at_bag(B,L), at_thing(T,L), outside(T)]. If O is ground we can define O.Rhs to be the set of $substates$ in the right hand sides of $O.nec$.

The definition of the move operator illustrates the specification of a conditional transition. In the example the conditional transition asserts that if any 'thing' is at the same location as the bag (A) and is inside the bag then it changes state to being at location (B) the new location of the bag and remains inside the bag. Where there is more than one transition in a conditional section they form a disjunction. The move operator is defined as follows:

```
operator(move(X,A,B),
```

```
% prevail
[],
% necessary
[(bag,X,[at_bag(X,A),ne(A,B)]
         =>
        [at_bag(X,B)])
],
% conditional
[(thing,T,[at_thing(T,A),inside(T,X),
            fits_in(T,X)]
           =>
          [at_thing(T,B),inside(T,X),
            fits_in(T,X)])
])
```

3 The Graphplan System

Graphplan [2] has proved to be one of the fastest plan generation algorithms working with a traditional STRIPS-like planning representation. Since its introduction a number of authors have proposed amendments with a view to improving the efficiency of the algorithm further e.g. [5]. Here we give only a very brief review of the algorithm, given the amount of published literature already using it. Graphplan works by building a plan-graph representing all possible plans creatable from the initial state by application of the available operators. If we consider the set of propositions true in the initial state as being at level 1 in our plan-graph then at level 2 will exist the set of all operations that are applicable, i.e. have their preconditions fulfilled by the propositions of level 1. At level 3 will be the set of propositions made true by the application of the operators of level 2. This process continues by developing the graph in exactly the same manner to additional levels. In the developing graph we record the application of operators as links that connect the propositions of the adjacent odd numbered levels. This process of moving from one level of propositions to the next supported by the application of operators is augmented by the application to every proposition at level *n* with a special operator *no-op* that renders the proposition true at level *n + 2*. This forward development of the graph faces a problem in that clearly in all proposition levels other than level 1 there may be propositions that cannot be jointly true. In the briefcase world the bag 'briefcase' cannot be at home and at the office. Likewise in a link layer actions may be mutually exclusive. The actions of moving the briefcase home and the action of moving it to the office cannot be simultaneously undertaken. We think of each proposition level as recording what potentially might be true at the same instant. We think of each link layer as recording the operations that might consistently be applied in parallel or where no commitment to ordering is required. The inconsistencies within a layer are recorded within Graphplan by augmenting the graph further by noting these mutually exclusive relations both between operations in the link layers and by recording mutually exclusive relations at the proposition layers. The development of the graph in this way from one proposition layer to the next mediated by a link layer constitutes the forwards phase of Graphplan.

To complete Graphplan a backwards search phase is required to find if a legal plan that satisfies the goal condition has been generated. This backwards phase is undertaken after the generation of each proposition layer, and starts by first searching the new proposition layer to see if all the propositions of the goal state are supported at this level. If they are not then the backward phase can be terminated and the next forwards phase started. If the goals are all present then the goal propositions must be checked to ensure that there are no recorded mutual exclusions between any of them. The backwards phase continues finding a set of operations that support these propositions and are themselves mutually consistent then recursively checking the preconditions of those operations in the same manner at the level two below. This process continues until we have regressed to the propositions of level 1 which by definition must be consistent with one another. If at any layer we find that the chosen set of operators are not mutually consistent then we must backtrack and see if an alternative set of operations can be chosen to support the same set of propositions. In this way Graphplan will continue interleaving its forwards and backwards phases to find an optimally parallel short legal plan, if one exists.

3.1 Conditional Effects in Graphplan

Since the original description of Graphplan a number of authors have described algorithms to extend Graphplan to allow the processing of conditional effects [7, 1]. In their paper Anderson Smith and Weld argue that the relatively simple approach of expanding the conditional effects section into all combinations of possible groundings is not feasible in cases dealing with significant numbers of possible groundings. They propose instead what they call a 'factored expansion approach'. Their approach requires that a operator with conditional effects be composed of clauses, one for the non-conditional component of the STRIPS operator and one for each grounding of the conditional clause conjoined with the non conditional element. The resulting $move - briefcase$ operator with the cheque and the dictionary is as follows:

```
move-briefcase (?loc ?new)
  :effect
    (when (and (at briefcase ?loc)
               (location ?new)
               (not (= ?loc ?new)))
          (and (at briefcase ?new)
               (not (at briefcase ?loc))))
    (when (and (at briefcase ?loc)
               (location ?new)
               (not (= ?loc ?new))
               (in cheque briefcase))
          (and (at cheque ?new)
               (not (at cheque ?loc))))
    (when (and (at briefcase ?loc)
               (location ?new)
               (not (= ?loc ?new))
               (in dictionary briefcase))
          (and (at dictionary ?new)
               (not (at dictionary ?loc))))
```

A consequence of this approach is that each of the elements becomes a semi-independent rule which can be fired separately which results in a requirement for more complex processing of mutex relations during the search phases of the Graphplan algorithm.

The approach we take in OCLGraph is similar to that of both Anderson et al and Koehler et al [1, 7], in that when we ground the operators the result will have one clause (object transition) in the conditional effects section for each object for which the grounding of the conditional effects clause is consistent with the necessary and prevailing sections of the operator. The growth of the number of clauses in the conditional effects section as a result of grounding is linear. It is bounded by the number of objects in the problem domain of the

correct object sort. We will delay further discussion until we have presented the OCLGraph algorithm.

4 The Object Graph

4.1 OCL Input

We will assume that the domain model is input using a restricted from of OCL to coincide with the input language specified in reference [2], but extended to deal with conditional effects. In particular, OCL operator schemas are translated to a ground set. The conditional element is expanded to include all consistent groundings of the conditional element. During the grounding which is done as a preprocessor step, static predicates are used to ensure consistent groundings. For example the static information about which objects fit in the briefcase and which objects fit in the suitcase is used to ensure that a conditional transition for moving the 'suit' which does not fit in the briefcase is not generated. The ground operators to move the briefcase in a world containing a cheque a dictionary and a suit from home to the office expands to:

```
operator(move(briefcase, home, office),
  % Prevail
  [],
  % Necessary
  [(bag,briefcase,
        [at_bag(briefcase, home)]
        =>
        [at_bag(briefcase, office)])
   ],
   % Conditional
   [
   (thing,cheque,
        [at_thing(cheque, home),
         inside(cheque, briefcase)]
        =>
        [at_thing(cheque, office),
         inside(cheque, briefcase)]),
   (thing,dictionary,
        [at_thing(dictionary, home),
         inside(dictionary, briefcase)]
        =>
        [at_thing(dictionary, office),
         inside(dictionary, briefcase)])
   ] )
```

A problem input to OCLGraph is defined by an initial state (a total mapping between dynamic object identifiers and substates) and a goal condition (a mapping between object identifiers and ground substate expressions).

4.2 Building Up the Graph

We build an 'OCL-graph' in the spirit of Graphplan by first substituting the idea of a proposition level with what we call an 'object level', defined as a (total) mapping (called $level(n)$ where n is odd) between the set of object identifiers $O\text{-}ids$ and the partitioned set of all possible substates for that object:

level(n) : O-ids $\Rightarrow$ Table

where Table is a set of substates partitioned by the substate class definitions. The intuitive idea is that if an object situation (s,i,ss) is potentially reachable at level n through the execution of operators then ss will be somewhere in the (partitioned) set 'level(n)[i]'.

Two immediate consequences of this representation are that:

(a) The size of every object level in a plan graph is always fixed as the number of objects in the initial state, although the size of the range sets of this map generally increases to the point where all legal substates for the objects, as defined in the substate class definition, are in the range.

(b) In a literal-based Graphplan any subset of the propositions at each propositional level can form a goal set which is potentially satisfiable. For example in the briefcase world, the set {in_thing(cheque,briefcase), at_thing(cheque,home),outside(cheque)} would be acceptable in principle, but would be found to be inconsistent through operator back chaining. OCL restricts goal sets to a set of $legal$ object expressions - hence the above expression would not be allowed as the cheque's substate expression is not well formed (it is not a specialisation of either one of *thing's* two substate classes).

4.2.1 Example

To create level(n+2) from level(n), we copy over the old mapping (this parallels the use of 'no-ops' in reference [2]) and add new substates to level(n+2)'s range if they are created by operator application at level(n+1). Consider the briefcase world with only two locations (home (h) and office (o)) and two things (cheque (c) and dictionary (d)). In the initial state b, c and d are all at home, c is inside b and d is not inside a bag. Then the development from the initial state in level 1 to level 3 is as follows:

```
level(1)[c] =
   {partition 1:[at_thing(c,h),inside(c,b)}
level(1)[d] =
   {partition 1:[at_thing(d,h),outside(d)]}
level(1)[b] =
   {partition 1:[at_bag(b,h)]}

level(3)[c] =
   {partition 1:[at_thing(c,h),inside(c,b)],
                 [at_thing(c,o),inside(c,b)],
    partition 2:[at_thing(c,h),outside(c)]}
level(3)[d] =
   {partition 1:[at_thing(d,h),outside(d)],
    partition 2:[at_thing(d,h),inside(d,b)]}
level(3)[b] =
   {partition 1:[at_bag(b,h)],
                 [at_bag(b,o)]}
```

The operators applicable at level 2 are take_out(c,b), put_in(d,b), and move(b,h,o), with the conditional effect of moving the cheque from home to the office.

4.3 Links

Definition of 'contains' If SE is a set of ground object expressions, n is odd, then $contains(level(n), SE)$ is true iff for each (s,i,se) in SE, there is a substate ss $\in$ level(n)[i] such that se $\subseteq$ ss.

An operator is applicable to level(n) if contains(level(n),O.Pre) is true, where O.Pre are the operator's preconditions as defined above. For example, contains(level(3),[at_bag(b,o)]) is true. Note that O.Pre excludes any elements for the operators conditional effects.

Definition of Links Assume operator O is applicable at level(n). Then a link $lk(O, i, ss, mode)$ is stored in level(n+1) if either (a) O $changes$ i's substate to ss or (b) (s,i,se) $\in$ O.prev, ss $\in$ level(n)[i] and se $\subseteq$ ss or (c) O is a no-op preserving ss from level(n)[i] to level(n+2)[i]. We record $mode$ as either 'change', 'prevail' or 'no-op' depending on each of the cases (a) - (c).

In the running example we therefore store the following:

```
level(3)[c] =
   {partition 1:[at_thing(c,h),outside(c)],
    partition 2:[at_thing(c,h),inside(c,b)]}
level(3)[d] =
   {partition 1:[at_thing(d,h),outside(d)],
    partition 2:[at_thing(d,h),inside(d,b)]}
level(3)[b] =
   {partition 1:[at_bag(b,h)],
                 [at_bag(b,o)]}

level(2) =
{lk(no-op-1,c,
      [at_thing(c,h),inside(c,b)], no-op),
 lk(take_out(c,b),c,
      [at_thing(c,h),outside(c)],change),
 lk(take_out(c,b),b,[at_bag(b,h)],prevail),
 lk(no-op-2,d,
      [at_thing(d,h),outside(d)],no-op),
 lk(put_in(d,b),d,
      [at_thing(d,h),inside(d,b)],change),
 lk(put_in(d,b),b,[at_bag(b,h)],prevail),
 lk(no-op-3,b,[at_bag(b,h)],no-op),
 lk(move(b,h,o),b,[at_bag(b,o)],change)}
```

To process the conditional effects in the forwards phase of the algorithm, new links and object substates at level $n + 2$ are added as follows:

Definition of Conditional Links For each conditional effect transition cc in an applicable operator O at level $n + 1$, if
$\exists$ ss $\in$ level(n): cc.lhs $\subseteq$ ss
then add $cc.rhs$ to level $n + 2$ and add link lk(O,i,cc.rhs,cond) to level(n+1) (hence the link here is labelled 'cond').

For the briefcase this adds a new substate to $level(3)[c]$ and adds a new link to record the application of the effect as follows:

```
level(3)[c] =
   {partition 1:[at_thing(c,h),outside(c)],
    partition 2:[at_thing(c,h),inside(c,b)],
                [at_thing(c,o),inside(c,b)]}

lk(move(b,h,o),c,
   [at_thing(c,o),inside(c,b)],cond)}.
```

We have applied one of the conditional elements in the 'move' operator. In applying such conditional elements we only consider operators that have already been applied, that is operators that have their prevailing and necessary preconditions fulfilled at that level, these operators already have their necessary effects and links recorded as described above.

4.4 Mutual Exclusions in OCLGraph

The forward development of the plan graph spreads in the manner described above. It is checked, however, by the use of mutual exclusion conditions on both operators and substates in the object levels. Blum and Furst's 'Interference' statement ([2], section 2.2) is paraphrased as follows: 'If either of actions O1 and O2 deletes a precondition or Add-Effect of the other, they are mutually exclusive at that level. Secondly if actions O1 and O2 have preconditions which are recorded as mutually exclusive then they are mutually exclusive' The idea is then to check each operator at each level against all the others, resulting in a set of binary mutual exclusions.

We exploit the structure of OCL to give the following definition:

General Rule for Operator Mutex Formation For each object identifier i in the object level(n+2), two distinct operators O1 and O2 are mutually exclusive if lk(O1,i,ss1,mode1) and lk(O2,i,ss2,mode2) are links recorded in level(n+1).

In other words, if two operators support the same object then they are mutually exclusive to one another.

The rationale is as follows: if operators O1 and O2 change or rely on the same object being in a particular substate, then they would in general interfere with each other. There are, however, some exceptions to the general rule above. Firstly, if ss1 = ss2, then at least one of mode1 or mode2 must be "change". If ss1 = ss2 and no mode is "change", then it does not follow that O1 and O2 are mutually exclusive. In practice we say O1 and O2 conflict if there is a reference to different substates of the same object in the preconditions or necessary effects of the operators. Secondly if mode1 = cond or mode2 = cond then we do not add the conflict at this stage as the conditional effect may not be used in the final plan even though the operator is. We do not in the forwards development of the graph detect if the firing of one element in an operator will force the firing of another. This is contrary to the practice of [1].

The case made by [1] for the need to record such induced mutexes derives from two cases.

- If two components of an operator are such that the preconditions of one of the components cannot be logically met without meeting the preconditions of the second component then we need to record that component one will be mutexed with all the operators component two is mutex with.
- The second case is harder to paraphrase but essentially if component one can fire and due to absence of other information the only way component two could fail to fire is if component one did not fire then again we can deduce that one forces two and should be mutexed with the operators two is mutexed with.

In OCL the first of the cases cannot arise as each element of an operator will refer to a different object and hence the preconditions for a conditional transition to fire cannot be contained in the other elements of the operator. The second case can arise. For example in the briefcase world if at level one the cheque is inside the briefcase, and we move it, then the cheque will also move. There is no other possibility as no other possible state of the cheque is recorded at this level. At later levels other states of the cheque will also be recorded and hence there will not be the same guarantee that moving the briefcase moves the cheque.

We could search for such cases but they are just a special case of a conditional effect being forced as a result of the interplay of the preconditions of a set of operators at a given level. We could not deal

with the general case in the forward phase of graph development as the set of operators will be dependent on the choices made in identifying a candidate valid plan. We therefore leave the backwards search phase of the planner to take care of potential conflicts arising from such conditional effects.

Employing this method to the example above, the mutexes turn out to be:

```
mutex(2) = {
{ no-op-1, take_out(c,b)},
{ no-op-2, put_in(d,b)},
{ move(b,h,o), no-op-3 },
{ move(b,h,o), take_out(c,b) },
{ move(b,h,o), put_in(d,b) } }
```

The mutex that we miss by delaying consideration of conditional effects is `{move(b,h,o), no-op-1}` . That is we cannot move the briefcase from home to the office with the cheque inside and simultaneous leave the cheque inside the briefcase at home. Note that the exceptions to the general mutex rule rule collapses the mutex formed by considering the 'briefcase' to binary mutexes.

Mutual exclusion conditions on object levels: In the original Graphplan description, two propositions p1 and p2 were mutually exclusive if all operators creating proposition p1 were exclusive of operators for creating p2. In the OCL formulation, we have two object class expressions (s,i,ce1) and (s,j,ce2) are mutually exclusive if

- for i <> j, for any operator O that supports ce1 and operator O1 that supports ce2, O and O1 are mutually exclusive (as defined by the binary mutexes described above)
- for i = j, ce1 and ce2 cannot be satisfied by a common ground substate

The first condition is similar to the original idea. The second arises from the fact that an object cannot be in two substates at the same level.

5 The OCL-graph Algorithm

5.1 Forwards Phase

Figure 1 shows the overall algorithm. Line 1 initialises the first level in the plan graph using the initial state. If the goals are not trivially achieved (Line 3), the algorithm builds two new levels, a new object level (n+2) and a link level (n+1) First in Line 7 the object states of level n are copied to level n+2 and the no-ops links added (note each no-op is given a unique identifier no-op-X). Following the addition of the no-ops, the code in the internal loop (Lines 8 to 23) applies the domain operators initially without reference to their conditional effects and the new object level is augmented and appropriate links added (lines 11,to 15). Following the application of an operator each transition of the operator's conditional effects is considered and if its preconditions are met and do not conflict with the preconditions of the prevail and necessary section it is applied and the appropriate substates and links added to the corresponding levels. (lines 16 to 20) After the loop adding all new substates to level n+2 and all links to level n+1 completes, operator mutex sets are built and added to level $n+1$ in Line 24.

5.2 Backwards Phase

Figure 2 details the definitions of 'ACHIEVE' which has overall control of the backwards search for a valid plan. ACHIEVE searches for

algorithm OCL-graph
In O-ids : Object identifiers; I : O-ids ⇒ Substates, Ops : Ground Operators, G : Goals
Out P : Parallel Plan
Types level(n) (n odd) is a map O-ids ⇒ Table, level(n) (n even) is a set of links
Types mutex(n) is a set of operator sets

1. ∀ i ∈ O-ids: level(1)[i] = {I[i]}
2. n := 1;
3. ACHIEVE(G,1, P) ;
4. while (P = null) do
5. level(n+2) := level(n);
6. links(n+1) := { };mutex(n+1) = { };
7. ∀ i ∈ O-ids: ∀ ss ∈ level(n+2)[i] :
   add lk(no-op-X, i, ss, no-op) to level(n+1);
8. ∀ O ∈ Ops do:
9. if contains(level(n), O.Pre) then
10. if not MUTEX(O.Pre,n) then
11. ∀(s,i,ss) ∈ O.rhs: add ss to level(n+2)[i],
12. add lk(O,i,ss,change) to level(n+1);
13. ∀ (s,i,se) ∈ O.prev:
14. if se ⊆ ss & ss ∈ level(n+2)[i]
15. then add lk(O,i,ss,prevail) to level(n+1);
16. ∀ cc ∈ O.cond:
17. if contains(level(n),cc.lhs) &
    not MUTEX(Pre ∩ cc.lhs)
18. then add cc.rhs to level(n+2)[cc.i];
19. add lk(O,cc.i,cc.rhs,cond), to level(n+1);
20. end if
21. end if
22. end if
23. end for;
24. calcuate all binary mutexes and add to mutex(n+1)
25. n := n+2;
26. if contains(G,level(n)) then ACHIEVE(G, n, P);
27. end while
28. end.

Figure 1. An Outline of the Object-Graph Planning Algorithm

a consistent operator set Y to achieve the goal set G, and if it finds one first calls COND_PRECONDITIONS to determine which conditional effects of the operators in set Y are required to achieve G and adds the preconditions of those elements to the necessary and prevailing preconditions of the operators Y. ACHIEVE then recursively calls itself at level(n-2) with the set of preconditions of Y as the new goals to achieve. The definition of consistent in Line 6 is left open ended, and depends on whether mutexes are stored concerning substates, as well as checking to see whether a goal expression is well formed with respect to the substate class definitions. The current OCL-graph implementation does not memoize substate mutexes, but this is a subject for on-going research.

The strategy for selecting conditional effects is shown in Figure 3. In line 1 we determine which substate expressions of the Goal state have not been supported by the necessary or $no-op$ effects of the chosen operator set O, these are the substate expressions that must

```
procedure ACHIEVE(SS : set of substate expressions, n : odd
integer, P : plan );
Global levels, mutexes
Out a parallel plan P;

1. if n = 1 & contains(level(1), SS) then P = { }
2. elseif n = 1 and not contains(level(1),SS) then
3.    P = null
4. else
5.    P' = null;
6.    choose Y := a consistent set of operators that
      achieve a set of substates containing SS;
7.    while(Y <> null & P' = null) do
8.       Y' := union of all the operators necessary
         and prevailing preconditions in Y;
9.       Y" := {};
10.      while(Y" <> null & P' = null) do
11.         Y" := COND_PRECONDITIONS(SS,Y,n)
12.         if Y" <> null then
13.            ACHIEVE({Y'∪Y"},n-2,P')
14.      end while
15.      if not(P' = null) then
16.         P := append(P',Y)
17.      else
18.         systematically generate another choice for Y
19.      end if
20.   end while
21. end if
22. end.
```

Figure 2. Achieve Procedure for the Object-Graph Algorithm

be supported by the conditional effects. Line 2 selects a set of those transitions from the conditional effects of the operators O that satisfies the unfulfilled goals SS'. The procedure then iterates on the selected set of transitions if any to check their consistency. To check the consistency of a selection we first determine those conditional effect transitions contained in the operator set O which are not needed to support the goal (Spare). We then check that the preconditions of each of the 'Required' transitions is consistent with the main preconditions of the selected operator set O and that none of the Spare conditional effects would if they are fired by the preconditions already required conflict with the outcomes of the operators selected. If these conditions are met we have successfully chosen the conditional effects needed and simple return them otherwise we must see if an alternative set of conditional effect elements can be generated to meet the requirement.

The primary method for determining that a set of object states are consistent is the function 'MUTEX'. Figure 4 It does the checking very simply, by trying to find a set of consistent operators at the level below which add these substate expressions. Operator sets are consistent if no two operators in the set are mutually exclusive.

```
function COND_PRECONDITIONS(SS : goal set,
O : operator set, n : odd integer): set of substate expressions
Global levels,mutexes

1. SS' := {SS - {se : se ∈ O.rhs}};
2. Required := a set of conditional elements from
      O.COND that achieve a set of substates
      containing SS';
3. while Required <> null do
4.    Spare := {O.COND - Required};
5.    if not MUTEX({O.Pre ∪ Required.lhs},n) &
6.       ∀ cc ∈ Spare
7.          if cc.lhs satisfied in {O.Pre ∪ Required.lhs} then
               not cc.rhs conflicts with
               {O.rhs ∪ Required.rhs}
8.    then
9.       return Required;
10.   else
11.      Required := choose new set from O.COND
            that achieves the set of substates containing SS'
12.   end if
13. end while
14. return null;
15. end.
```

Figure 3. Selection Conditional Effect Elements for Plan Inclusion

```
function MUTEX(SS : set of substate expressions, n : odd integer):
boolean
Global levels, mutexes

1. if n = 1 & contains(level(1), SS) then
2.    MUTEX := false
3. else if n = 1 and not(I contains SS) then
4.    MUTEX := true
5. else if ∃ Y, a set of operators that achieve a set
      of substates containing SS, and
      no two operators in Y are mutexed then
6.    MUTEX := false
7. else MUTEX := true
8. end.
```

Figure 4. Detecting mutex relations in a set of Object Substates

6 Implementation

To try and establish the benefits of using OCL in a Graphplan like algorithm we initially created two separate distinct planners, implemented in Sicstus Prolog. The first, though it could process OCL descriptions of planning domains, made no attempt to benefit from the structure. Rather it was used to simply extract the elements of

the standard STRIPS style operators. This implementation was designed to form our base measure for conducting experiments in an attempt to investigate the advantages in utilising the structures inherent in OCL. We refer to this implementation of Graphplan as 'vanilla' Graphplan. The second implementation "OCLGraph" tries to fully exploit the structure of OCL. Though the results derived from these implementations were encouraging they are not reliable as an objective comparison of the underlying algorithms. To try and rectify this position we are currently developing a new version of OCLGraph in Lisp with a view to comparing its performance against other public domain versions of Graphplan. In particular "Sensory GraphPlan" [3] is ideal for this purpose as it provides a clean faithful implementation of Graphplan but also encorporates extensions to deal with conditional effects.

7 Empirical Results

Tests have been carried out on a number of the standard 'toy' domains, such as the Rocket, Robot and Flat Tyre world and the Briefcase world for conditional effects. The tests have involved comparing times of the vanilla version of Graphplan against the Prolog OCL version and comparing times of "Sensory" Graphplan with the Lisp version of OCLGraph.

The results of our tests are problematic in that a clear speed up is indicated when we compare the Prolog version of OCLGraph with our own vanilla version of Graphplan and on the most favourable problems this can be by as much as 100 fold. However these results are not replicated when we compare the Lisp version of OCLGraph with "Sensory Graphplan". In this case the results mostly indicate no significant difference but with some problems our OCLGraph performed slower by an order of magnitude. The problems where our software was performing worse than "Sensory Graphplan" were in examples where the goal state was generated at a relatively early level in the progress of the forwards phase of the algorithm but where a legal plan was not generated until several levels deeper into the graph expansion. This happens in examples such as tasks in the "Flat Tyre World" where the tools have to be returned to the "boot", their initial state, after being used to fix the tyre.

Clearly more work needs to be done with the code to ensure the faithfulness in the implementations. That the code produces the expected results is not sufficient guarantee that the algorithms are accurately implemented. The relatively poor results with our Lisp implementation may result from fewer mutexes being recorded in the forward search than should be. A problem of this nature would degrade performance but not prevent the eventual production of the correct answer.

8 Analysis

We would expect the performance of a Graphplan-like algorithm to be influenced primarily by control of the branching factor of the graph. The factors influencing the degree of branching are:

- The most obvious factor is the creation of mutex relations during the forwards phase of the graph expansion. The identification of substate class definitions at domain design time in OCL provides the algorithm builder with inexpensive methods for identifying mutex relations between predicates referring to the same object. However though OCL makes it easier to find mutexes it is not clear that mutexes are found that would not be found in standard Graphplan.
- In OCL versions of Graphplan the branching factor relating to an action should be reduced by the fact that propositions are grouped together into substate descriptions of specific objects. In a backwards search for a legal plan if we have selected an action as a candidate for inclusion in the plan we need to check the producers of each of the substates of the objects referred to in the action. In the non-OCL version we need to check the producers of each of the separate propositions referred to in the preconditions of the action. For example if we want to add to a candidate plan a *move* action in the "Briefcase World" in the OCL version we need to consider the producer of the briefcase and that of each object in the briefcase. In a situation with two objects in the briefcase we have three producers to include at the next lower level. In the non-OCL version we need to check the producer of the proposition describing the *briefcase's* location and for each object in the briefcase we need to check the producer of the proposition stating the location of the object and the proposition stating that the object is in the briefcase and potentially the static predicate that the object fits in the briefcase. Again with two objects in the briefcase we need to check one producer for the briefcase but three for each of the two objects in the briefcase. The fan out from the non-OCL version thus seems significantly greater than the fan out from the OCL version.
- Another way the OCL formulation of Graphplan helps control search manifests itself when dealing with conditional effects. As detailed in [7], the backwards search needs to take account of the unwanted firing of conditional effects, which would interfere with the achievement of the plan goals. In OCLGraph as objects can only be in one substate at a time we don't need to both check that (a) an object is in some desired substate, required for the achievement of the goal and (b) that it is not in some other substate to prevent the firing of an unwanted conditional effect.
- The grouping of propositions into descriptions of object states as done in OCL in some circumstances increases the branching factor of the graph by introducing more action instantiations at a given level of the graph than is done in the standard version of the algorithm. This problem is best described with the aid of an example. Consider enhancing the description of objects in the "Briefcase World" to allow us to record states of the objects. For example we might want to record whether or not the suit is "clean" or "dirty". In which case we might augment state descriptions of objects by introducing a predicate $state(Object, Property)$. Such predicates are then added to all state descriptions of objects. Such a change would have no impact on how the move operator is described in the STRIPS style representations used by the standard Graphplan algorithm nor would it make any difference to the number of move operations applicable at any level in the graph. The state of any of the objects, as described above would be unaffected by the *move* operator and would be deemed to persist. In an OCL version of the *move* operator there would be both a change to its representation and potentially to the number of applicable move operations at a level in the graph. In the representation of the OCL version of the operator we would need to refer to the state of the object being moved as the right-hand-side of a transition must fully specify the resulting state of the object concerned. This would have the consequence that at some levels of the graph we would generate one move operator to move a "clean" suit from the office to home and another operator to move a "dirty" suit. There is no such duplication in the traditional Graphplan algorithm.

It would seem then that the extra structure of the OCL representation pulls in both directions. Both allows us in some circumstances to reduce the branching factor of the graph and in other circumstances increases it.

9 Conclusions

In this paper we have illustrated how a graph-based algorithm can be extended to more structured representations of planning domains. We have argued that there is potential for efficiency gains though there are also threats. Our analysis and experimentation is not yet at a sufficiently mature stage to accurately determine the extent of the trade off between the competing factors. Our design of the Object-Graph algorithm has uncovered various ways in which the extra information content of OCL can be used to make the graph-based algorithm more efficient but we have not been able to remove the potential threats to efficiency, though this may be possible in a hierarchical formulation of Graphplan.

There are many avenues for future work. First we would like to extend the experimental base to cover cases with a greater diversity of graph sizes, and to experiment with more interesting domains possessing more structure. Secondly, there is a need to analyse the computational complexity of the OCL-based algorithm in greater depth, and compare it with the original. Thirdly, we need to extend the algorithm to be able to accept the full OCL language, and to improve the algorithm so that it uses yet more of the extra information given in an OCL model. For example, domain invariants typically found in an OCL model often read as mutex constraints on a pair of substates. Finally, improvements to the basic algorithm such as dependency directed backtracking [4] have not been implemented but there is no reason to expect that they would not be equally applicable to our version of the algorithm.

REFERENCES

[1] Corin R. Anderson, David E. Smith, and Daniel S. Weld, 'Conditional Effects in Graphplan', in *The Fourth International Conference on Artificial Intelligence Planning Systems*, (1998).

[2] A. L. Blum and M. L. Furst, 'Fast planning through Planning Graph Analysis', *Artificial Intelligence*, **90**, 281–300, (1997).

[3] C.Anderson, D.E.Smith, and D.Weld, *Sensory Graphplan* , http://www.cs.washington.edu/research/projects/ai/www /projects/Sensory_Graphplan/, 2000.

[4] S. Kambhampati, 'On the relations between intelligent backtracking and explanation-based learning in planning and constraint satisfactions', *Artificial Intelligence*, **105**, (1998).

[5] S. Kambhampati, E. Parker, and E. Lambrecht, 'Understanding and Extending Graphplan', in *Proceedings of the 4th European Conference on Planning*, (1997).

[6] D. E. Kitchin, *Object-Centred Generative Planning*, Ph.D. dissertation, School of Computing and Mathematics, University of Huddersfield, forthcoming,2000.

[7] J. Koehler, B. Nebel, J. Hoffmann, and Y. Dimopoulos, 'Extending Planning Graphs to an ADL Subset', in *Proceedings of the 4th European Conference on Planning*, (1997).

[8] D. Liu and T.L.McCluskey, 'The OCL Language Manual, Version 1.2', Technical report, Department of Computing Science, University of Huddersfield , (2000).

[9] T. L. McCluskey, 'Object Transition Sequences: A New Form of Abstraction for HTN Planners', in *Proceedings of the 5th International Conference on Artificial Intelligence Planning and Scheduling Systems (aips-2000)* , (2000).

[10] T. L. McCluskey and J. M. Porteous, 'Engineering and Compiling Planning Domain Models to Promote Validity and Efficiency', *Artificial Intelligence*, **95**, 1–65, (1997).

[11] B. Pell N. Muscettola, P. P. Nayak and B. C. Williams, 'Remote Agent: To Boldly Go Where No AI System Has Gone Before', *Artificial Intelligence*, **103(1-2)**, 5–48, (1998).

[12] J. M. Porteous, *Compilation-Based Performance Improvement for Generative Planners*, Ph.D. dissertation, Department of Computer Science, The City University, 1993.

Dynamic Scheduling of Progressive Processing Plans

Shlomo Zilberstein[1] and **Abdel-Illah Mouaddib**[2] and **Andrew Arnt**[3]

Abstract. Progressive processing plans allow systems to tradeoff computational resources against the quality of service by specifying alternative ways in which to accomplish each step. When the structure of a plan is known in advance, it can be optimally scheduled by solving a corresponding Markov decision process. This paper extends this approach to dynamic scheduling of plans that can be constantly modified. We show how to construct an optimal meta-level controller for a single task and how to extend the solution to the case of multiple and dynamic tasks using the notion of an opportunity cost. Several fast approximation schemes for the opportunity cost are evaluated. The results provide an effective framework for managing computational resources in highly dynamic environments.

1 INTRODUCTION

This paper is concerned with dynamic scheduling of progressive processing task structures. In this framework, each task is mapped to a *progressive processing unit* (PRU) composed of a set of modules that can contribute to the quality of the result. The problem is to select at run-time the best subset of modules so as to maximize the quality of the result produced with limited computational resources.

While the general framework is extremely general, we focus in this paper on a particular information retrieval application. Information retrieval from a large collection involves uncertainty regarding the duration of the process and the quality of the result. In addition, there may be large variability in the number of requests that require a response at any given time. By taking a context dependent, dynamic approach to the problem we can significantly improve the average quality of service provided by such systems.

A typical search engine is composed of several information retrieval modules that perform such tasks as query formation, query optimization, query evaluation, precision improvement, recall improvement, clustering, and results visualization. For each one of these phases, there are a wide variety of techniques that have been developed in recent years [12]. Currently, search engines are built by choosing and integrating a fixed set of modules and techniques. The choices are made off-line by the designer of the system. This static approach excludes techniques that work well in special situations. In addition, current information retrieval systems are optimized for a particular load; they cannot respond dynamically to varying load, availability of computational resources, and to the specific characteristics of a given query.

The ability to dynamically adjust computational effort based on the availability of computational resources has been studied extensively by the AI community since the mid 1980's. These efforts have led to the development of a variety of techniques such as *anytime algorithms* [1, 13], *design-to-time* [2], *flexible computation* [5], *imprecise computation* [7], and *progressive reasoning* [8, 9].

In particular, the progressive processing approach offers a natural framework to describe the set of information retrieval techniques available to the system. Figure 1 shows a simple task structure whose input is a *query* composed of a list of keywords. The task structure has three processing *levels*. The first level includes three alternative techniques to improve the initial query: (a) scan the query using concept recognizers to identify company names, dates, locations, personal names, and so on; (b) examine the query for pairs of words that have high statistical likelihood of being related and enhance the query with that information; (c) perform part-of-speech analysis to identify noun phrases within the query. The second level includes two alternative techniques that can improve the query's recall ability by expanding it to include related words and phrases: (d) use of Local Context Analysis (LCA), a statistical method for expanding queries that relies upon in-context analysis of word co-occurrence; (e) use of InFinder, an association thesaurus that is faster than LCA and does not capture context as well. Finally, the third level performs the actual query evaluation and returns the results. Quality in this application is measured by the number of relevant documents within the top n documents retrieved (i.e., precision in the retrieved set).

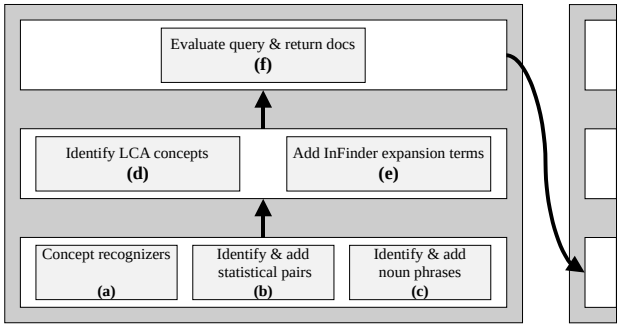


Figure 1. Illustration of a progressive processing task for an information retrieval search engine

This information retrieval application provides a good example of several fundamental issues:

1. Handling the *duration uncertainty* and *quality uncertainty* associated with each technique.
2. Handling the *dependency* of quality and duration on the quality of intermediate results.
3. Handling a rich task structure in which some levels include several *alternatives* or optional computational steps; optional steps can be skipped under time pressure, leading to direct evaluation of the

[1] Department of Computer Science, University of Massachusetts, Amherst, MA 01003, USA, email: zilberstein@cs.umass.edu
[2] CRIL-IUT de Lens-Université d'Artois, Rue de l'université, S. P. 16, 62307 Lens Cedex, France, email: mouaddib@cril.univ-artois.fr
[3] Department of Computer Science, University of Massachusetts, Amherst, MA 01003, USA, email: arnt@cs.umass.edu

input query.

4. Selecting the optimal set of retrieval techniques in a *dynamic* environment taking into account the entire set of queries waiting for execution.

The rest of this paper offers an efficient solution to the meta-level control problem. Section 2 gives a formal definition of the problem. We then solve the problem in two steps. In Section 3, we develop an optimal solution for a single PRU, ignoring the fact that additional tasks are waiting for processing. Section 4 shows how to handle multiple PRUs using the approach developed in Section 3 and summarizing the effect of the waiting requests using the notion of an opportunity cost. In section 5 we address the issue of reactive control in a highly dynamic environment by estimating the opportunity cost and pre-compiling the control policies. We conclude with a summary of the results and a brief description of related work.

2 THE META-LEVEL CONTROL PROBLEM

This section describes formally the problem of meta-level control of the progressive processing model. Each information retrieval request is mapped to a task structure described below.

Definition 1 *A* **progressive processing unit** *(PRU) is composed of a sequence of processing levels,* $(l_1, l_2, \ldots, l_L)$*. The first level receives the input query and the last one produces the result.*

Definition 2 *Each processing level,* l_i*, is composed of a set of* p_i **alternative modules**, $\{m_i^1, m_i^2, \ldots, m_i^{p_i}\}$.

Each module can perform the logical function of level l_i, but it has different computational characteristics defined below.

Definition 3 *The* **module descriptor**, $P_i^j((q', \delta)|q)$*, of module* m_i^j *is the probability distribution of output quality and duration for a* given *input quality.*

Note that q is a discrete variable representing quality and δ is a discrete variable representing duration. The module descriptor specifies the probability that module m_i^j takes δ time units and returns a result of quality q' when the quality of the previously executed module is q. Module descriptors are similar to *conditional performance profiles* of anytime algorithms [13]. They are constructed empirically by collecting performance data for a sample set of inputs.

When the search engine responds to a particular request, it receives an immediate reward defined as follows.

Definition 4 *A* **time-dependent utility function**, $U(q, t)$*, measures the utility of a solution of quality* q *if it is returned* t *time units after the arrival time of the request.*

We assume that there is a given constant T such that $\forall q, t > T : U(q, t) = 0$. That is, responding to a request more than T time units after its arrival has no value.

Suppose that a system maintains a set of information retrieval requests, W, with arrival times $\{a_1, a_2, \ldots, a_n\}$. The set of requests is updated dynamically as new requests arrive. The system processes the requests in a first-in-first-out order using a progressive processing unit to handle each request.

Given a set of requests, the module descriptors of all the components of the progressive processing unit, and a time-dependent utility function, we define the following control problem.

Definition 5 *The* **reactive control problem** *is the problem of selecting a set of alternative modules so as to maximize the expected utility over the set of information retrieval requests.*

The meta-level control is "reactive" in the sense that we assume that the module selection mechanism is very fast, largely based on offline analysis of the problem. The rest of the paper provides a solution to this problem.

3 OPTIMAL CONTROL OF A SINGLE PRU

We begin with the problem of meta-level control of a single progressive processing unit corresponding to a single task. This problem can be formulated as a simple Markov decision process (MDP) with states representing the current state of the computation. The state includes the current level of the PRU, the quality produced so far, and the elapsed time since the arrival of the request. The rewards are defined by the utility of the solution which depends on both quality and time. The possible actions are to *execute* a module of the next processing level or to *skip* that processing level. The transition model is defined by the descriptor of the module selected for execution. The rest of this section gives a formal definition of the MDP and the reactive controller produced by solving it.

3.1 State representation

The execution of a single progressive processing unit, u, can be seen as an MDP with a finite set of states $\mathcal{S} = \{[l_i, q, t] | l_i \in u\} \bigcup \{[\text{failure}, t]\}$ where $0 \leq i \leq L$ indicates the last executed (or skipped) level, $0 \leq q \leq 1$ is the quality produced by the last executed module, and $0 \leq t \leq T$ is the elapsed time since the arrival time, a_u, of the request. Note that quality is discretized and normalized to be in the range $[0..1]$. All the intermediate modules use a uniform representation of input and output (a "query" in our application). Note also that T is the maximum delay after which we consider the response to be useless. When the system is in state $[l_i, q, t]$, one module of the i-th level has been executed. (The first level is $i = 1$; $i = 0$ is used to indicate the fact that no level has been executed.) The states $[\text{failure}, t]$ represent termination at time t without any useful result. We distinguish between different failure states because failure can occur before the deadline leaving some remaining time for the execution of other requests in the queue.

3.2 Transition model

The initial state of the MDP is $[l_0, q_{init}, t]$, where t is the elapsed time since the arrival of the request ($t =$ current time $- a_u$) and q_{init} is the initial quality of the request (0 in our application). The initial state indicates that the system is ready to start executing a module of the first level of the PRU. The terminal states are all the states of the form $[l_L, q, t]$ or $[failure, t]$. The former set represents finishing execution of the last level and the latter set represents failure. Other states such as $[l_i, q_{max}, t]$ (reaching maximal intermediate quality) or $[l_i, q, T]$ (reaching the deadline before the execution of the last level) are not considered terminal states. A terminal state can be reached from state $[l_i, q, T]$ by executing a series of *skip* actions until a failure state is reached. Similarly *skip* actions take the automaton from state $[l_i, q_{max}, t]$ to the last level because no *execute* action can improve the intermediate quality.

In every nonterminal state the possible actions are: $\mathbf{E}_{i+1}^j$ (execute the j-th module of the next level) and **S** (skip the next level). To

complete the transition model, we need to specify the probabilistic outcome of these actions. Equations 1-4 define the transition probabilities for a given nonterminal state $[l_i, q, t]$.

The **S** action is deterministic. It skips the next level without affecting the quality or elapsed time. (It can be implemented as an additional "dummy" module whose execution takes no time and has no effect on quality.)

$$Pr([l_{i+1}, q, t] \mid [l_i, q, t], \mathbf{S}) = 1$$
$$\text{when } 0 \leq i < L-1 \tag{1}$$

Skipping the last level results in failure.

$$Pr([failure, t] \mid [l_{L-1}, q, t], \mathbf{S}) = 1 \tag{2}$$

The $\mathbf{E}^j_{i+1}$ action is probabilistic. Duration and quality uncertainties define the new state. Equation 3 determines the transitions following successful execution and Equation 4 determines the transition to the failure state when the deadline, T, is reached.

$$Pr([l_{i+1}, q', t+\delta] \mid [l_i, q, t], \mathbf{E}^j_{i+1}) =$$
$$P^j_{i+1}((q', \delta)|q) \quad \text{when } t+\delta \leq T \tag{3}$$

$$Pr([failure, T] \mid [l_i, q, t], \mathbf{E}^j_{i+1}) =$$
$$\sum_{q', \delta > T-t} P^j_{i+1}((q', \delta)|q) \tag{4}$$

3.3 Rewards and the value function

Rewards are determined by the given time-dependent utility function applied to the final result (produced by the last level of the PRU). The utility depends on the quality of the result and the elapsed time. Keep in mind that in our application the intermediate results are useless and therefore have no direct rewards associated with them. We now define a value function (expected reward-to-go) over all states. The value of terminal states is defined as follows.

$$V([l_L, q, t]) = R(q, t) = U(q, t) \tag{5}$$

$$V([failure, t]) = R(0, t) = U(0, t) \tag{6}$$

The value of nonterminal states of the MDP is defined as follows.

$$V([l_i, q, t]) =$$
$$\max_a \begin{cases} V([l_{i+1}, q, t]) & \text{If } a = \mathbf{S}, 0 \leq i < L-1 \\ V([failure, t]) & \text{If } a = \mathbf{S}, i = L-1 \\ EV([l_i, q, t] \mid \mathbf{E}^j_{i+1}) & \text{If } a = \mathbf{E}^j_{i+1}, 0 < j \leq p_i \end{cases} \tag{7}$$

Such that $EV([l_i, q, t] \mid \mathbf{E}^j_{i+1}) =$

$$\sum_{q', \delta > T-t} P^j_{i+1}((q', \delta)|q)\, V([failure, T]) +$$
$$\sum_{q', \delta \leq T-t} P^j_{i+1}((q', \delta)|q)\, V([l_{i+1}, q', t+\delta])$$

The value function is defined as maximum over all actions with the top expression representing the value of a skip action for any level l_i such that $0 \leq i \leq L-1$, the middle expression representing the value of a skip action for level l_{L-1}, and the bottom expression representing the value of an execute action.

This concludes the definition of an MDP. This MDP is a finite-horizon MDP with no cycles. It can be solved easily using standard dynamic programming algorithms or using search algorithms such as AO*.

Theorem 1 *Given* one *progressive processing unit u and a time-dependent utility function $U(q, t)$, the optimal policy for the corresponding MDP is an optimal reactive control for u.*

Proof: Because there is a one-to-one correspondence between the reactive control problem and the MDP (including the fact that the PRU transition model satisfies the Markov assumption), and because of the optimality of the resulting policy, we conclude that it provides optimal reactive control for the progressive processing problem. □

3.4 Choice of unit resolution

The number of states of the MDP we must solve to control a single PRU is bounded by the product of the number of levels L, the maximum number of alternative modules per level $\max_i p_i$, the number of discrete quality levels, and the maximum execution time. While the maximum execution time can be quite large, the time unit used for the purpose of meta-level control is an arbitrary system parameter. A small time unit leads to a more effective control at the expense of a larger state-space. The choice of a unit of quality has a similar effect. These units introduce a tradeoff between the size of the policy and its effectiveness. We evaluate this tradeoff below be measuring the policy size and construction time for different unit sizes. For the sake of simplicity, the same unit reduction factor, u, is used for both time and quality.

In this experiment, quality (q) and time (t) have the following ranges:

$$0 \leq q \leq 100$$
$$0 \leq t \leq T$$

where T, the failure state deadline, is between 300 and 1000. The unit resolution, u, defines the number of base level units grouped together into a larger unit size. The following table shows the number of discrete states per each level of the MDP for $T = 300$ and $u = 1, 5, 10, 20, 40, 80$.

Table 1. States per level as a function of unit resolution

u	# of t values	# of q values	states per level
1	301	101	30401
5	61	21	1281
10	31	11	341
20	16	6	96
40	8	3	24
80	5	2	10

The experiments were conducted with five randomly generated PRUs for each of the four types described in the table below. Type A is representative of the characteristics of an actual information retrieval application, while the others are used to test scalability.

For each PRU, the corresponding MDP was solved using the above six different unit values. For each resolution u, an optimal policy

Table 2. States per level as a function of unit resolution

PRU type	L	Modules per level	T
A	3	6	300
B	3	15	300
C	3	6	1000
D	3	15	1000

P_u was constructed. The value of the initial state, $[l_0, q_0, t_0]$, for $u = 1$ represents the precise initial state expected value. The policy P_k was used to select actions in a simulation starting from the initial state of the MDP. The simulation traced the precise state (with units size $u = 1$) while selecting actions based on the *approximate* policy. This simulation was repeated 1000 times for each u, recording the returned value (reward) of the initial state. Finally, we computed the relative error between the value achieved using P_k and the exact value for each type of PRUs, using five random cases of each type. The results are summarized in the following table.

Table 3. The effect of unit resolution on policy value and construction time

PRU Type	Exp Value	u	Avg Const Time(H:M:S)	Avg Value	Avg % Error
A	36.597	1	1:28:07.624	36.488	-0.297
		5	0:00:10.474	36.591	-0.017
		10	0:00:00.897	35.905	-1.891
		20	0:00:00.093	35.499	-3.001
		40	0:00:00.011	34.789	-4.942
		80	0:00:00.004	28.397	-22.406
B	14.254	1	3:27:20.762	14.301	0.329
		5	0:00:25.289	14.188	-0.469
		10	0:00:02.109	13.303	-6.676
		20	0:00:00.223	10.219	-28.310
		40	0:00:00.027	4.449	-68.786
		80	0:00:00.008	-10.279	-172.112
C	30.429	1	4:16:10.096	30.074	-1.167
		5	0:00:30.377	30.067	-1.189
		10	0:00:02.492	29.981	-1.472
		20	0:00:00.026	29.137	-4.245
		40	0:00:00.031	22.781	-25.135
		80	0:00:00.008	19.436	-36.127
D	21.378	1	13:18:36.698	21.464	0.401
		5	0:01:35.471	21.039	-1.585
		10	0:00:07.781	21.191	-0.876
		20	0:00:00.805	14.678	-31.338
		40	0:00:00.094	15.604	-27.007
		80	0:00:00.023	14.084	-34.118

Several important observations can be made based on the above table. First, it confirms the intuition that the value of a policy degrades gracefully as the unit size increases. But more importantly, the table shows that a unit size of 10 leads to a dramatic reduction in policy construction time with only a small relative error. For example, for type A PRUs, the time reduction is from more than 88 minutes to less than 1 second. The loss of value is less than 2%. These results confirms the applicability of the approach to realistic problems by adopting a good unit resolution.

4 OPTIMAL CONTROL OF MULTIPLE UNITS USING OPPORTUNITY COST

Suppose now that we need to schedule the execution of multiple PRUs. We assume that there are $n + 1$ requests whose arrival times are $a_0 \leq a_1 \leq \ldots \leq a_n$. One approach to construct an optimal schedule is to generalize the solution presented in the previous section. We can construct a larger MDP for the combined sequential decision problem including the entire set of $n + 1$ PRUs. To do that, each state must also include i, the request number, leading to a general state represented as $[i, l, q, t]$. Note that t is the elapsed time since the arrival of the *first* request.

This rather complex MDP is still a finite-horizon MDP with no loops. Moreover, the only possible transitions between different PRUs are from a terminal state of one PRU to an initial state of a succeeding PRU. Therefore, we can solve this MDP by computing an optimal policy for the *last* PRU for any starting time between 0 and $T + a_{n+1} - a_0$, then use the value of its initial states to compute an optimal policy for the previous PRU and so on.

Theorem 2 *Given a set, W, of progressive processing units and a time-dependent utility function $U(q, t)$, the optimal policy for the corresponding MDP is an optimal reactive control for W.*

This is an obvious generalization of Theorem 1. The complete proof, by induction on the number of PRUs, is omitted.

We now show how to reformulate the effect of the remaining n requests on the execution of the first task. This reformulation preserves the optimality of the solution, but it suggests a more efficient control structure developed in Section 5.

Definition 6 *Let $V_i^*(t) = V([i, l_0, q_0, t])$ denote the expected value of the optimal policy for the last $n - i + 1$ PRUs.*

To compute the optimal policy for the i-th PRU, we can simply use the following reward function.

$$R_i(q, t) = U(q, t + a_0 - a_i) + V([i + 1, l_0, q_0, t]) \quad (8)$$

In other words, the reward for responding to the i-th request is composed of the immediate reward (defined by the time-dependent utility function) and the reward-to-go (defined by the remaining PRUs). Alternatively, the reward can be represented as follows.

$$R_i(q, t) = U(q, t + a_0 - a_i) + V_{i+1}^*(t) \quad (9)$$

Therefore, the best policy for the first PRU can be calculated if we use the following reward function for final states:

$$R_0(q, t) = U(q, t) + V_1^*(t) \quad (10)$$

Definition 7 *Let $OC(t) = V_1^*(0) - V_1^*(t)$ be the* **opportunity cost** *at time t.*

The opportunity cost measures the loss of expected value due to delay in the starting point of executing the last n tasks (all the tasks except the first one).

Definition 8 *Let the* **OC-policy** *for the first PRU be the policy computed with the following reward function:*

$$R(q, t) = U(q, t) - OC(t)$$

The OC-policy is the policy computed by deducting from the actual reward for the first task the opportunity cost of its execution time.

Theorem 3 *Controlling the first PRU using the OC-policy is optimal.*

Proof: From the definition of $OC(t)$ we get:

$$V_1^*(t) = V_1^*(0) - OC(t) \tag{11}$$

To compute the optimal schedule we need to use the reward function defined in Equation 9 that can be rewritten as follows.

$$R_0(q,t) = U(q,t) + V_1^*(0) - OC(t) \tag{12}$$

But this reward function is the same as the one used to construct the OC-policy, except for the added constant $V_1^*(0)$. Because adding a constant to a reward function does not affect the policy, the conditions of Theorem 2 are met and the resulting policy is optimal. □

Theorem 3 suggests an optimal approach to scheduling the entire $n+1$ requests by first using an OC-policy for the first request that takes into account the opportunity cost of the remaining n requests. Then the OC-policy for the second request is used taking into account the opportunity cost of the remaining $n-1$ tasks and so on. To be able to implement this approach we need to have the control policies readily available. This issue is addressed in the following section.

5 REACTIVE CONTROL BASED ON ESTIMATED OPPORTUNITY COST

In the previous section, we presented an optimal solution to the control problem of multiple progressive processing units without accounting for its computational complexity. In particular, the opportunity cost must be computed and revised quickly each time a new request arrives. Once the opportunity cost is revised, a new policy for the current PRU must be constructed. Finding the exact opportunity cost requires the construction of an optimal policy for the entire set of tasks. In practice, this may slow down the operation of the information retrieval search engine.

In order to provide an effective reactive controller for dynamic progressive processing, it is necessary to:

1. use a fast approximation scheme to estimate the opportunity cost; and
2. use pre-compiled policies for different opportunity cost functions.

The rest of this section explains this method in more detail.

5.1 Estimating the Opportunity Cost

The opportunity cost is defined in terms of the function V_1^* which represents the value of an optimal policy for the remaining tasks in the queue. Thus, it can be estimated by approximating this function.

5.1.1 Naïve approximation

A naïve approach to approximating the cumulative value of the remaining tasks is to add the value of each task *without* taking into account the opportunity cost. In this calculation, the start time of each task is the *expected* end time of the previous one. The following set of equations summarizes this approximation scheme.

$$\begin{aligned}
&V_1^*(t) \simeq V([l_0, q_0, t + a_0 - a_1]) + V_2^*(t + \tau_1) \\
&\vdots \\
&V_i^*(t) \simeq V([l_0, q_0, t + a_0 - a_i]) + V_{i+1}^*(t + \textstyle\sum_{j=1}^{j=i} \tau_j) \\
&\vdots \\
&V_n^*(t) = V([l_0, q_0, t + a_0 - a_n])
\end{aligned} \tag{13}$$

where $V[l,q,t]$ is the value function defined in Section 3 for a single PRU. Therefore, V_1^* can be approximated as follows.

$$V_1^*(t) \simeq \sum_{i=1}^{i=n} V[l_0, q_0, t + a_0 + (\sum_{j<i} \tau_j) - a_i] \tag{14}$$

The expected duration of task i, τ_i, depends on the duration of the previous tasks. Let $\tau(d)$ be the expected duration of the optimal single-PRU policy when starting at time d (relative to the arrival time of the request). Then τ_i is computed using τ with the expected starting time of task i relative to *its* arrival time.

$$\begin{aligned}
&\tau_0 = 0 \\
&\tau_i = \tau(t + a_0 + (\textstyle\sum_{j<i} \tau_j) - a_i)
\end{aligned} \tag{15}$$

The function τ (expected duration) can be computed for any finite-horizon MDP once the optimal policy is available by simply using durations as rewards. The function can be computed once off-line, making it easy to revise the opportunity cost when a new request is added.

5.1.2 Learning an approximate opportunity cost function

Another approach is to estimate the opportunity cost using some features that characterize the remaining PRUs in the queue. Using a data set of pre-computed opportunity costs for many different queues, we can use these features to quickly approximate the opportunity cost for the current queue. The features used is our experiment are:

1. The total number of PRUs in the queue.
2. The average waiting time of a PRU in the queue.

We performed some experiments to determine the effectiveness of this approach. The dataset of queues was generated using a simple model of query arrival time (a random number between 0 and 3 requests arrive over a period of ten time units). The exact opportunity cost was computed at each of the time units for 100 randomly generated queues.

Two different estimation methods have been tested. The first was the k-Nearest-Neighbor algorithm, where we use a Euclidean distance metric on the features to determine the k closest data queues to the test queue. The estimated opportunity cost for each time unit is then the average of the opportunity costs over the k data queues for that time unit.

The second method was kernel regression, where we give all of the data queues a weight based on their feature-based Euclidean distance to the test queue. The weighing function for data queue i is as follows.

$$w_i = e^{\frac{-D(i,test)^2}{W}} \tag{16}$$

Where $D(x_i, test)$ is the distance between i and the test queue, and W is the kernel width. The estimated opportunity cost for time unit t is then the weighted average over all of the data queues.

$$est\_oc_t = \frac{\sum w_i oc_{t_i}}{\sum w_i} \tag{17}$$

To determine the effectiveness of each of these methods we used leave-one-out cross validation. In L-O-O CV, for each data queue i, we omit i from the data set and then use our approximation method to estimate the opportunity costs for i. We then compute the error e_i

between the estimated values and the actual values for i. Two different methods for computing e_i were tested.
Method 1:

$$e_i = \left(\sum_{t=0}^{10} \frac{|oc_{t_{actual}} - oc_{t_{est}}|}{oc_{t_{actual}}} \right) / 10 \quad (18)$$

Method 2:

$$e_i = \frac{\sum_{t=0}^{10} |oc_{t_{actual}} - oc_{t_{est}}|}{\sum_{t=0}^{10} oc_{t_{actual}}} \quad (19)$$

Method 1 averages together the fraction error from each time step, while Method 2 finds the error of the sum of all the opportunity costs. Then, the total cross validation error is the average of all of the e_i.

Figures 2 and 3 show that an acceptable error of less than 0.1 is achieved using 1 Nearest Neighbor or Kernel Regression with a very small width. An estimate with such a small error suits our needs.

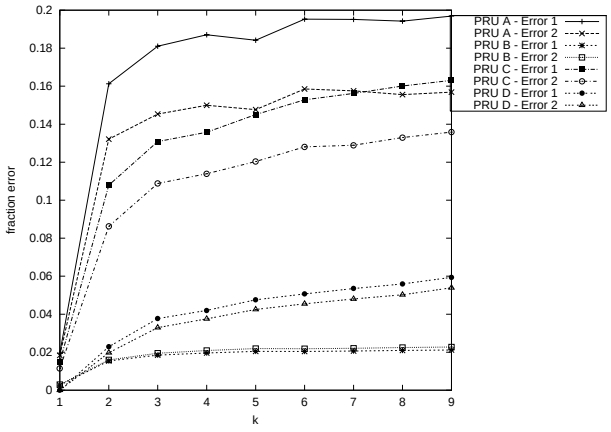


Figure 2. Leave one out cross validation error for k nearest neighbor

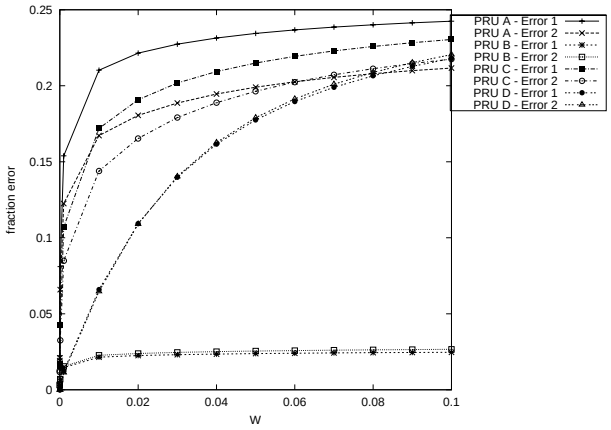


Figure 3. Leave one out cross validation error for kernel regression with width W

5.1.3 Comparison of opportunity cost approximation methods

We now compare the performance of the naïve method for opportunity cost approximation against the best feature-based function approximation technique, 1-Nearest-Neighbor, described above. Performance using no opportunity cost is given for comparison. This is based on an optimal policy for a single task, ignoring the entire queue of requests. Unlike the naïve method, in this case the the reward function does not take into account the reward-to-go.

To perform this comparison, 50 different PRU arrival queues were randomly generated for each of the four PRU types described in Table 2. They were generated by having a random number between 0 and 3 requests arrive at each time unit over a period of ten time units (using a unit size of 10 seconds). Note that all the PRUs in a given queue correspond to the same task structure. For each arrival queue, we estimated $OC(t)$ at each of the possible arrival times using both estimation methods. We then computed the actual opportunity cost. Table 4 gives the average relative error for each of the methods. As expected, 1NN generally outperforms the naïve method.

We also observed how often actions chosen by the estimated OC policy differed from those specified by the optimal policy. These values are also given in Table 4. We see that both of the estimation methods perform very well, with the 1NN method actually generating a policy identical to the optimal for PRUs of type A. Ignoring the opportunity cost leads to a large action error (up to 35%). It is interesting to note that in PRUs of type D, the action error is small for all three approaches. The large number of alternatives and high level of uncertainty about duration make the value of the second-best action closer to the value of the best action. Note also that in this case the naïve method provides the more accurate estimate of OC, but it also leads to larger action selection. A possible explanation is that while the estimate is more accurate in general, it is less accurate for some critical cases in which a small error makes a difference in action selection.

Table 4. Comparison of OC approximation methods

PRU type	OC Est Method	Est OC Error	Action Error
A	None	N/A	20.345
	Naïve	34.102	2.014
	1NN	7.178	0.0
B	None	N/A	18.422
	Naïve	10.550	5.586
	1NN	2.813	4.678
C	None	N/A	35.185
	Naïve	6.042	0.971
	1NN	2.668	0.233
D	None	N/A	1.453
	Naïve	1.142	1.302
	1NN	2.255	0.376

5.2 Pre-compiled control policies

To make the meta-level control truly reactive for large task structures, one may want to avoid computing a new policy (for a single PRU) each time the opportunity cost is revised. To avoid this, the space of opportunity cost can be divided into a small set of regions representing typical situations. For example, there could be just three regions that capture *low*, *medium*, and *high* loads. For each region, an optimal policy would be computed off-line and stored in a library. At run-time, the system will first estimate the opportunity cost and then use the appropriate pre-compiled policy from the library. These policies remain valid as long as the overall task structure and the utility function are fixed. Because the dependency of the control decisions

on the opportunity cost is monotonic (higher costs imply less time for execution), we anticipate that a small set of classes that correspond to *qualitatively* different action selection will be sufficient.

Another advantage of the use of pre-compiled policies is the ability to react quickly to dynamic changes. Control policies can be switched *during* the execution of a single request if the opportunity cost changes. This is possible because the policies share the same state space.

6 CONCLUSION

We present an innovative approach to meta-level control of progressive processing based on reformulating it as a Markov decision problem. It is shown that an optimal policy for a set of tasks can be constructed by controlling a single PRU, taking into account the opportunity cost of the remaining tasks. To apply this model to control the operation of an information retrieval search engine, a fast approximation of the opportunity cost is developed. Finally, a highly reactive controller is described that uses a library of pre-compiled control policies to operate in a dynamic environment.

A less complex model of progressive processing that relies on heuristic scheduling has been developed [9]. The task structure, however, is limited to a linear set of levels with one module per level and no quality uncertainty or quality dependency. The heuristic scheduler is fast, but it cannot solve the more complex task structure presented in this paper and it does not provide optimal control. Heuristic scheduling of computational tasks has also been studied by Garvey and Lesser [1993] for the *design-to-time* problem-solving framework. The latter framework represents explicitly non-local interactions between sub-tasks.

The progressive processing framework relates to a large body of work within the systems community on *imprecise computation* [7]. Each task in that model is decomposed into a *mandatory* subtask and an *optional* subtask. A variety of scheduling algorithms have been developed for imprecise computation under different assumptions about the optional part. Our model allows for a richer representation of quality and duration uncertainty and quality dependency. Unlike imprecise computation, the schedule constructed by the MDP scheduler is a *conditional schedule*; the selection of modules is conditioned on the *actual* execution time and outcome of previous modules.

The application of dynamic programming to solve meta-level control problems have been previously used by Hansen and Zilberstein [1996] to control interruptible anytime algorithms. Optimal monitoring of progressive processing tasks using a corresponding MDP has been studied by Mouaddib and Zilberstein [1998] with respect to a simpler task structure and without the notion of quality uncertainty and quality dependency.

The notion of *opportunity cost* is borrowed from economics. It has been used previously in meta-level reasoning by Russell and Wefald [1991]. Horvitz [1997] uses a similar notion to develop a model of *continual computation* in which idle time is used to solve anticipated future problems.

The use of pre-compiled control policies to construct a highly reactive real-time system has been studied by several researchers. For example, Greenwald and Dean [1998] show how a real-time avionics control system can use a library of schedules that cover all possible situations. Each schedule is conditioned on the state of the flight operation.

In collaboration with the Information Retrieval Center at UMass we are currently developing the stochastic module descriptors for the components of the search engine. By definition, IR tasks involve large collections and a substantial amount of test data allowing us to test the applicability and scalability of this resource-bounded reasoning technique.

ACKNOWLEDGMENTS

We thank James Allan and Victor Lavrenko for their contribution to the problem formulation and to the construction of the information retrieval testbed.

This work was supported in part by the National Science Foundation under grants No. IRI-9624992, IIS-9907331, and INT-9612092, by the GanymedeII Project of Plan Etat/Nord-Pas-De-Calais, and by IUT de Lens.

REFERENCES

[1] T. Dean and M. Boddy, An analysis of time-dependent planning, *Seventh National Conference on Artificial Intelligence*, 49–54, 1988.

[2] A. Garvey and V. Lesser, Design-to-time real-time scheduling, *IEEE Transactions on Systems, Man, and Cybernetics*, 23(6):1491–1502, 1993.

[3] L. Greenwald and T. Dean, A conditional scheduling approach to designing real-time systems, *AI Planning systems*, 1229–1234, 1998.

[4] E.A. Hansen and S. Zilberstein, Monitoring the progress of anytime problem-solving, *Thirteenth National Conference on Artificial Intelligence*, 1229–1234, 1996.

[5] E. Horvitz, Reasoning under varying and uncertain resource constraints, *Seventh National Conference on Artificial Intelligence*, 111–116, 1988.

[6] E. Horvitz, Models of continual computation, *Fourteenth National Conference on Artificial Intelligence*, 286–293, 1997.

[7] J. Liu, K. Lin, W. Shih, A. Yu, J. Chung, and W. Zao, Algorithms for scheduling imprecise computations, *IEEE Transactions on Computers*, 24(5):58–68, 1991.

[8] A.-I. Mouaddib, *Contribution au raisonnement progressif et temps réel dans un univers multi-agents*, PhD thesis, University of Nancy I, (in French), 1993.

[9] A.-I. Mouaddib and S. Zilberstein, Handling duration uncertainty in meta-level control of progressive reasoning, *Fifteenth International Joint Conference on Artificial Intelligence*, 1201–1206, 1997.

[10] A.-I. Mouaddib and S. Zilberstein, Optimal scheduling of dynamic progressive processing, *Thirteenth Biennial European Conference on Artificial Intelligence*, 449–503, 1998.

[11] S. Russell and E. Wefald, *Do the Right Thing: Studies in Limited Rationality*, MIT Press, 1991.

[12] K.S. Jones and P. Willett (eds.), *Readings in Information Retrieval*, Morgan Kaufmann Publishers, 1997.

[13] S. Zilberstein and S. Russell, Optimal composition of real-time systems, *Artificial Intelligence* 82(1-2):181–213, 1996.

Authors Index