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.

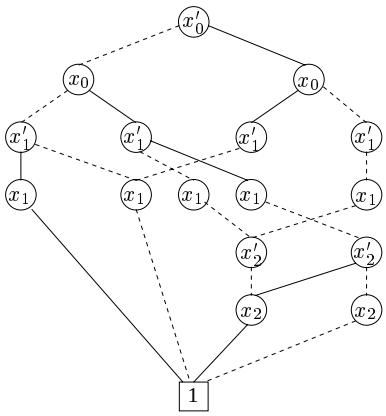

Figure 1. The transition relation for the example problem. For the sake of clarity, the *false* sink has been omitted. Dashed lines and solid lines indicate edges labeled *false* and *true*, respectively.

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 $\leftarrow \phi_{\{s\}}$
  do
    Succ $\leftarrow \exists x'\ (Open(x') \wedge T(x', x))$
    Open $\leftarrow$ *Succ*
  while (*Open* $\wedge\ \phi_G \equiv 0$)

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.

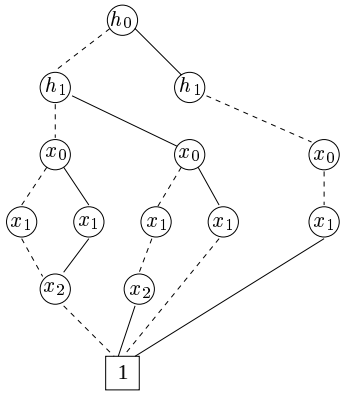

Figure 6. The BDD representation for the heuristic function in the example problem. In this case the individual pattern values have been maximized.

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.

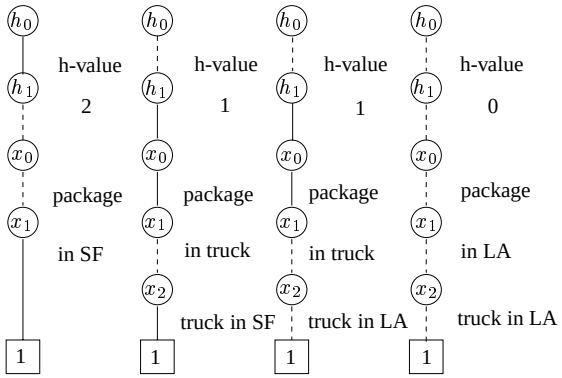

Figure 9. Backward exploration of the example problem in *Pure* BDDA*. In each iteration step the BDD *Min* with associated h-value is shown. Note that when using forward set simplification these BDDs additionally correspond to a snapshot of the priority queue *Open*.

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).