Accelerating Heuristic Search in Spatial Domains

Stefan Edelkamp[1], Shahid Jabbar[2], and Thomas Willhalm[3]

[1] Fachbereich Informatik
Universität Dortmund
Baroper Str. 301
44221 Dortmund
stefan.edelkamp@cs.uni-dortmund.de
[2] Institut für Informatik
Universität Freiburg
Georges-Köhler-Allee 51
79110 Freiburg
jabbar@informatik.uni-freiburg.de
[3] Universität Karlsruhe
Institut für Logik, Komplexität
und Deduktionssysteme
76128 Karlsruhe
willhalm@ira.uni-karlsruhe.de

Abstract. This paper exploits the spatial representation of state space problem graphs to preprocess and enhance heuristic search engines. It combines classical AI exploration with computational geometry.
Our case study considers trajectories in the plane. The application domain is general route planning, independent to the underlying vehicle model. The implemented target system is located on a server and answers client's shortest path queries with respect to a set of global positioning system traces.
A graph is constructed from the traces and is compressed while retaining the original information for unfolding the resulting shortest paths. The search space is pruned by the database entries concisely representing all shortest paths that start with a given edge. The latter algorithm is suited for a server application that has to serve a large number of queries from several different clients.
The sudden changes in the topology of graph can affect the pre-computed database entries. This paper discusses some models to incorporate the changes in the graph and possible solutions.

1 Introduction

Heuristic search is a fundamental concepts in AI to enhance solution path finding in implicit problem graphs [29]. It applies to both implicit and explicit state spaces. In this context a state space problem is a tuple $\mathcal{P} =< \mathcal{S}, \mathcal{O}, \mathcal{I}, \mathcal{G} >$, where $\mathcal{S}$ is the set of states, $\mathcal{I} \in \mathcal{S}$ is the initial state, $\mathcal{G} \subseteq \mathcal{S}$ is the set of goal states, and $\mathcal{O}$ is the set of operators that transform states into states. A heuristic h is a mapping from $\mathcal{S}$ into the set of non-negative real numbers and estimates the costs of the remaining exploration problem. For each final state we expect estimate zero.

Route planning is one prototypical example for explicit heuristic search. It refers to the situation where the shortest path between the current location u and target location t has to be found on a map that is available in form of a directed weighted state space graph $G = (V, E, w)$. We restrict to the single-source single-target shortest path problem, since the minimal distance to more than one goal node can be obtained by merging them into one. Searching for such a single-pair shortest path is often performed with a run of Dijkstra's algorithm. In this case, the exploration process can be terminated, once the goal node has been found. In spatial domains, like route planning problems in the plane, one can include estimated distance information to accelerate solution finding.

A suitable heuristic in the route planning domain is $h(u) = || \ t - u \ ||_2$, where $|| \cdot ||_2$ is the Euclidean norm. In some cases (like navigating in the alpes) including the third dimension in the estimate can be advantageous. Locations closer to the goal have smaller priority as those that are farther apart. Adding the accumulated distance $w(p_u) = \sum_{i=1}^{l} w(v_{i-1}, v_i)$ of path $p = \langle s = v_0, \ldots, v_l = u \rangle$ from s to u to the estimate $h(u)$ results in an approximation $\overline{w}(p_u)$ of the minimal cost from s to t. To update the total path estimate when moving from u to v we have $\overline{w}(p_v) = w(p_v) + h(v) = w(p_u)+w(u,v)+h(v) = \overline{w}(p_u)-h(u)+w(u,v)+h(v)$. Since $w(u, v)$ is larger than the straight distance between u and v, that is $w(u, v) \geq ||u - v||_2$, the heuristic $h(u) = || \ t - u \ ||_2$ is *consistent*, since $h(u) = ||t - u||_2 \leq ||t - v||_2 + ||u - v||_2 \leq h(v) + w(u, v)$ by the triangle inequality for vector norms. On every path from the initial state to a goal node, the accumulated estimate values telescope. Since the goal cost estimate is zero, the node priorities at termination time of the original and refined algorithm are the same, assuming an additional offset of $h(s)$ at the start node s in the latter, often denoted as A* [29]. Subsequently, at least for consistent estimates, A* (without any re-opening strategy) is complete and optimal.

It is possible to derive consistent distance metrics for many AI domains. SAT-problems [19] with n variables, for example, can be represented in n dimensional space and when variable flips were used to produce successor configurations, the Hamming distance is an appropriate estimate. Each configuration in the $(n^2 - 1)$-puzzle can be evaluated by the Manhattan distance, for which two adjacent positions differ by value 1 [31]. A related application area is the search for multiple sequence alignments, which also induce weighted graphs according to edit distances [39]. The metric to considered here is the exact distance for the problem in a lower dimension, i.e., when aligning less sequences.

This paper presents a case study of route planning with respect to time-stamped sequences of points, so-called traces. From different application areas that produce such points we select global position traces produced by GPS (Global Positioning System) receivers. It has yet to be shown if the techniques we consider are also of help in other AI search domains. There is some hope, since the heuristic itself defines a metric to be exploited by geometric algorithms.

We have structured the paper as follows. First we introduce the application domain of GPS trace navigation and how the data is gathered. Then we turn to the graph construction process by applying efficient geometric algorithms to the set of GPS traces. We then discuss the graph compression process where we merge all the nodes having an *in-degree* and *out-degree* of 1. The graph is annotated to accelerate shortest path queries

in the form of Euclidean distance heuristics, assisted by geometric shortest path pruning information. In Section 8, we discuss the effects of sudden change in the topology of the graph due to disasters like road accidents. All route-finding algorithms preserve optimal routes. In the experimental section we evaluate our implementation on a few sets of collected GPS data. Last but not least we present related, current and future work.

2 Data Gathering

Route planning is one of the most important industrial application areas of AI search [10]. Current technology like hand-held computers, car navigation and GPS positioning systems ask for a suitable combination of mobile computing and course selection for moving objects, but most maps come on external storage devices and are by far larger than main memory capacity. This is especially true for on-board and hand-held computer systems. Furthermore, most available digital maps are expensive, since exhibiting and processing road information e.g. by surveying methods or by digitizing satellite images is very costly. Maps are likely to be inaccurate and to contain systematic errors in the input sources or inference procedures. It is costly to keep map information up-to-date, since road geometry continuously changes over time. Maps contain information on road classes and travel distances only, which is often not sufficient to infer travel time. In rush hours or on public holidays, the time needed for driving deviates significantly from the one assuming usual travel speed. In some regions of the world digital maps are not available at all.

Therefore, we develop a domain-independent route planning system that takes a set of traces as an input, pre-computes an according graph structure and quickly answers different on-line queries. According to the navigational task, there are different opportunities to track and process data. Although the algorithms we present are independent from the application area, we briefly discuss the case of cycling to impose low cost and high mobility requirements like small size and low battery consumption.

In Figure 1 you see a moderately equipped bike to perform advanced mobile data gathering. At the handlebars we installed a GPS receiver[4] and an advanced cycling computer[5]. The GPS device features a small base map, waypoint and route following options, as well as up- and downloading of data. Raw GPS information is sent by the receiver in a frequency of one signal per second to its serial port. This data is stored in a palm-top device with a free-ware program[6] that was developed to memorize flight data. The memorized traces are transfered to a PC by a simple terminal program[7]. A gender changer attaches the two serial cables of GPS and palm-top. Since the frequency of data storage in the cycling computer is low (one data point per 20s), we additionally experimented with a simple wheel magnet directly linked to a palm-top to be processed in a shareware program[8] that memorizes ticks each second.

[4] Garmin Venture, `www.garmin.com`
[5] Ciclosport HAC-4 Plus, `www.ciclosport.de`
[6] GPS logger, `www.palmflying.com/glogger.html`
[7] Serialterm, `www.comp.lancs.ac.uk/~albrecht/sw`
[8] Bikini, `home.swipnet.se/~w-51358/pilot`

Fig. 1. A bike equipped to gather GPS data.

Besides the recording of current velocity through incoming wheel-turn ticks, the chosen cycling computer includes an integrated altimeter and a heart pulse counter. Stored heart pulse data allows to generate personalized maps and annotated routes probably cautioning the cyclist to expect stress and to make a rest. To have accurate GPS independent orientation, an electronic (gyro)compass is needed.

Bike navigation is only one of the vehicle routing aspects we look at. For proper car navigation, inertial information can be accessed on the internal electronic bus and for the design of autonomous mobile systems, efficient outdoor GPS path (re-)planning and path following is of growing importance. In fact, the algorithms we develop are generic in the sense that they apply to any kind of motion in the physical world that can access GPS information. In near future by the UMTS technology we expect all different devices to be merged into one. Assistance in the form of cruise, steering and break control imposes certain security problems, so that car manufacturers often omit automated control facilities even if the according technology might be applicable.

Searching timed traces in the plane has also broad applicability in robotics. Suppose that in some explanatory phase, robots with laser range finder generate a series of scans that are matched against each other and to build a positioning trace, of where the robot is. Storing trace graphs are by far smaller than the storing scans.

3 Refining and Reducing Data

An immediate refinement for raw data is differential information through geodetic reference points that is processed with respect to the current position. This additional source

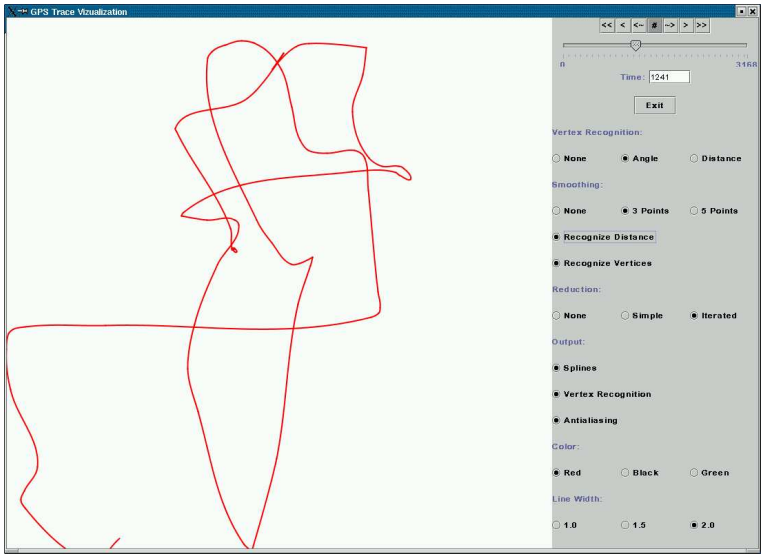

Fig. 2. GPS trace visualization.

is standardized and can be included to the positioning data even by low end devices. However, the low-cost access to portable high precision reference data is limited. Existing systems[9], which send encoded signals via radio or GSM signals or high-end GPS devices. Kalman filters can used to detect outliers in the GPS data. We refer the reader to the literature for autonomous robotics, e.g. [15].

For further data reduction, we apply the Douglas-Peuker *geometric rounding* algorithm [16]. The method was developed to reduce the number of points to represent a digitized curve from maps and photographs. It considers a simple trace of $n+1$ points $\{p_0, \ldots, p_n\}$ in the plane that form a polygonal chain and asks for an approximating chain with fewer line segments. It is best described recursively: to approximate the chain from point p_i to p_j the algorithm starts with segment $p_i p_j$. If the farthest vertex from this segment has a distance smaller than a given threshold θ, then the algorithm accepts this approximation. Otherwise, it splits the chain at this vertex and recursively approximate the two pieces. The $O(n \log n)$ algorithm takes advantage of the fact that splitting vertices are to be located on the convex hull of the enclosed points[10].

Smoothing GPS data is required, since the data contains inherent vibrations. Programming languages like JAVA already provide anti-aliasing as a system call. Figure 2 displays GPS data with splines and iterated data reduction. First experiments confirm the data in the context of handwriting that considerably savings can be achieved with iterated point removal in long traces without removing the main characteristics of the trace.

[9] cf. SAPOS, `www.sapos.de`

[10] See `www.mpi-sb.mpg.de/~mehlhorn/SelectedTopics02/GeometricRounding/GeometricRounding.ps` for a recent overview on geometric rounding techniques.

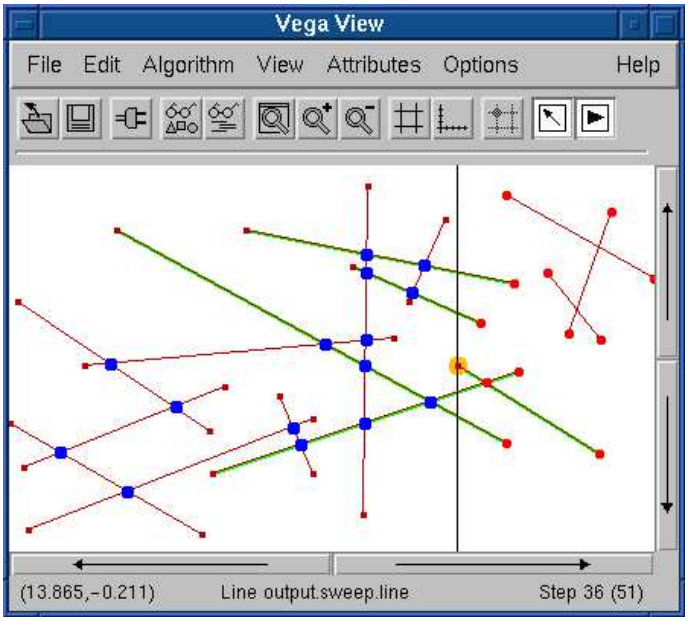

Fig. 3. GPS trace intersection.

4 Graph Construction

The problem graph consists of the set of traces together with the according intersections. To compute the graph, the sweep-line segment intersection algorithm of [2] has been adapted. Figure 3 gives an animated example of the algorithm. In difference to the original algorithm, the generated graph is weighted and directed. At the intersections the newly generated edges inherit direction, distance and time from the original data points. The algorithm runs in $O((n+k)\log n)$ time, with n being the number of data points and k being the number of intersections. The lower bound of the problem's complexity is $\Omega(n \log n + k)$ and the first step to improve time performance was $O(n \log^2 n / \log\log n + k)$ [3]. The first $O(n \log n + k)$ algorithm [4] used $O(n+k)$ storage. The $O(n \log n + k)$ algorithm with $O(n)$ space is due to [1].

The graph G can be defined as a 4-tuple, $G = (V, E, d, b)$, where V is the set of vertices and E is the set of edges. The function $d : E \rightarrow I\!R$ assigns with each edge, the Euclidean distance between the source and target vertex of the edge. The function $b : V \rightarrow Time$ gives the temporal information about the vertex, precisely - the date and time when that point was visited.

5 Graph Compression

Once the graph has been constructed we observe that there are a large number of nodes having in-degree = out-degree = 1. These nodes do not create any *choice* in the exploration phase of a search algorithm. This motivates to apply compression scheme, where

the edges incident to these nodes can be merged into a single edge that starts and ends at an intersection node or at a node representing the start or end point of a trace. The weight of that new edge will be made equal to the sum of the weights of the intermediate edges incident to the nodes of degree 2.

The edges incident to the nodes of degree 2 cannot be deleted since this will loose the original layout of the graph, which is important for two reasons: First, the user is not concerned with the compressed graph and is interested in having the answer to his/her query in the form of GPS points defining the exact path on the streets. Second, in the compressed graph it will be possible only to start and end the search from the intersection nodes or the end points of traces. This leads us to retain the information about the original graph and maintain a two-layered graph i.e. the compressed graph on top of the original graph. This approach for compressing the graph reduces the space complexity from $O(n+k)$ to $O(l+k)$, where l is the number of traces and $l << n$.

The compressed graph $G_c = (V_c, E_c, d_c, b)$ of $G = (V, E, d, b)$ can be defined as a layer on top of the original graph G where $V_c = \{v_c \in V \mid (indeg(v_c) = 1$ and $outdeg(v_c) = 0)$ or $(indeg(v_c) = 0$ and $outdeg(v_c) = 1)$ or $(indeg(v_c) > 2)$ or $(outdeg(v_c) > 2)\}$, E_c are the new compressed edges, and d_c - the weight function - can be defined for an edge $e_c = (u_c, v_c) \in E_c$ as the sum of the weights of all the intermediate edges $(u_c = v_1, v_2), (v_2, v_3), \ldots, (v_{k-1}, v_k = v_c)$. i.e., $d_c(u_c, v_c) = \sum_{i=1}^{k-1} d(v_i, v_{i+1})$. Note that due to the compression there is no change in the timing information b. More precisely, the compression is a surjective mapping $\phi : V \rightarrow V_c$, so that $e_c = (u_c, v_c) \in E_c$ if there exists a path $p =< u, v_1, \ldots, v_k, v >$ and $indeg(v_i) = outdeg(v_i) = 1$ with $\phi(u) = u_c$ and $\phi(v) = v_c$.

In practice, compression is performed through a linear time algorithm that scans all the nodes of the graph and creates an edge $e = (u, w)$ if there exists the edges $e_1 = (u, v)$ and $e_2 = (v, w)$ with $indeg(v) = outdeg(v) = 1$. If e_1 or e_2 are themselves results of some merging process, then they are deleted otherwise they are marked hidden in order to be restored later.

If $e_c = (u_c, v_c)$ is an edge in the compressed graph then both u_c and v_c have degrees larger than 2 (except when they are the start or end point of a trace). In order to decompress e_c, we need a handle to the correct hidden edge from u_c than can take us to v_c. For this purpose, during compression, we maintain a mapping $\psi : E_c \rightarrow E$ that when given $e_c = (u_c, v_c)$ returns the first hidden edge in the path from u_c to v_c.

Due to the compression, we can only start the search from the intersecting vertices or the start and end points of a trace. So, whenever a query $q = (s, t)$ is posed to the system, it is transformed into $q_c = (s_c, t_c)$, where $s_c \in V_c$ is the nearest reachable vertex in G_c from s. Intersection s_c can be found out by a mere walk starting from the single outgoing edge from s until we reach a compressed vertex. If $s \notin V$, we first use *nearest neighbor search* to find out the vertex $v \in V$ closest to s. The compressed vertex s_c is then searched with reference to v. The new target, t_c can then be defined conversely, except that it corresponds to the nearest compressed vertex while *backtracking* from t.

This leads to the partition of a path Π in to a sequence of three sub-paths as $< \Pi_{pre}$, Π_{inter}, $\Pi_{post} >$, where Π_{pre} - the *prefix* path - is the path from s to s_c. Π_{post} - the *postfix* path is the path from t_c to t. The sub-path Π_{inter} is the shortest path from s_c to

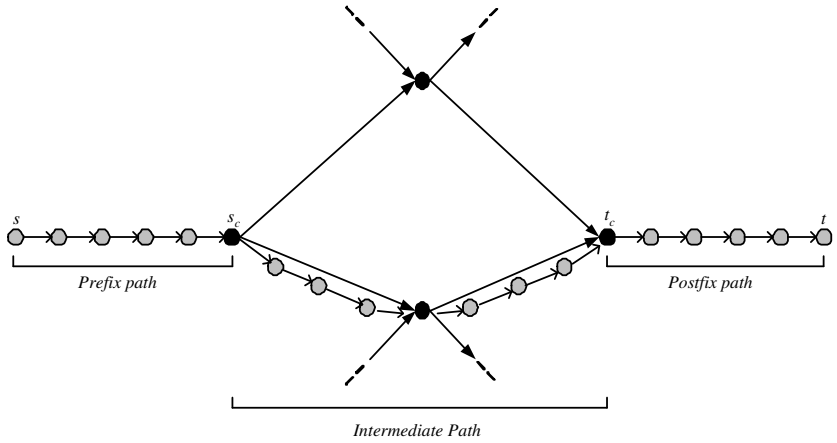

Fig. 4. Partition of a path.

t_c. Note that there can exists only single Π_{pre} and Π_{post} with all the intermediate nodes having in-degree = out-degree = 1 and hence no ambiguity can arise. (See Figure 4)

6 Querying

If the user's query points, s and t, do not belong to the graph G, a nearest neighbor search is done to find the nearest points to s and t in G. For a set of queries, this is best accomplished by an assisting point localization structure that contains nearest neighbor information. The apparently suited data structure is the Voronoi diagram [36], which for n points can be constructed in $O(n \log n)$ time. The structure consists of Voronoi regions $V(p)$ for each point p, which in turn are fixed by the intersections of all $n-1$ half-planes according to the bisectors to the other points. All points in the interior of $V(p)$ are nearer to p than to any other point in the point set. Voronoi diagrams are also often attributed to [7]. There are two main algorithms that meet the given running time: the sweep- (or beach-) line algorithm [12] and the divide-and-conquer strategy of separating paths [14]. Figure 5 depicts an example of the Voronoi diagram of a small set of points.

Building a query structure on top of the Voronoi diagram is available in $O(\log n)$ time by hierarchical subdivision [20]. Probably the best practical option to generate the Voronoi diagram is via its geometric dual - the Delaunay triangulation - since it yields a simple randomized strategy in expected time $O(n \log n)$ with expected optimal storage requirements [13]. If we take the set of segments instead of the set of points to be queried, extended Voronoi diagrams, e.g. [5] or trapezoidal maps are appropriate. An efficient randomized algorithm computes the trapezoidal map and the according search structure simultaneously runs in $O(n \log n)$ expected time and $O(n)$ expected space [28]. The expected query time is $O(\log n)$.

Let us briefly pause to reflect the situation in more generally embedded state spaces. Let $X \neq \emptyset$ and $d : X \times X \rightarrow I\!R$. We call (X, d) a metric space if $i)$ $d(x, y) \geq 0$, $ii)$

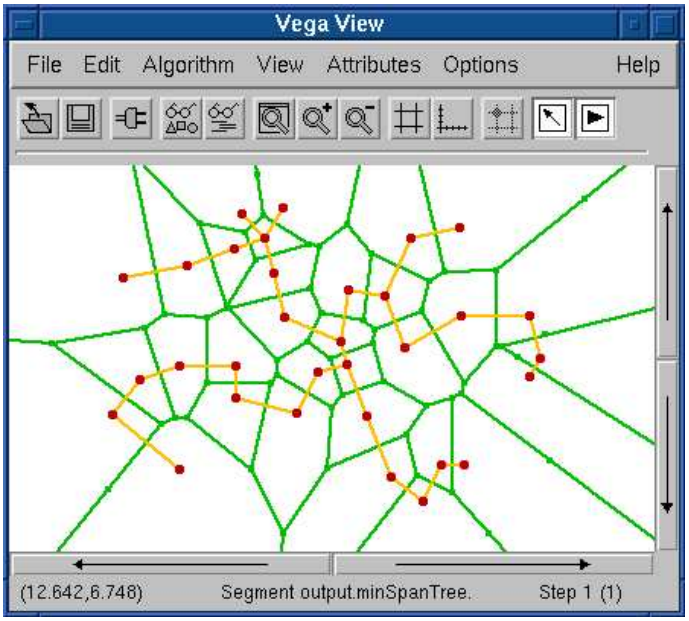

Fig. 5. Voronoi Diagram for GPS point localization.

$d(x,y) = 0 \iff x = y$, $iii)$ $d(x,y) = d(y,x)$, and $iv)$ $d(x,y) \leq d(x,z) + d(z,y)$. Metric spaces induce a spatial structure, which can in turn be exploited by geometric algorithms. If X is a vector space and $||\cdot||$ is a mapping $X \rightarrow I\!R$ then we call $||\cdot||$ a norm for X if $i)$ $||x|| \geq 0$, $ii)$ $||x|| = 0 \iff x = 0$, $iii)$ $||\alpha x|| = |\alpha|||x||$, and $iv)$ $||x+y|| \leq ||x|| + ||y||$. If X has norm $||\cdot||$ then $d(x,y) = ||x-y||$ is a metric for X. In this paper we take the Euclidean metric as default notation. One more general metric is the L_p metric, for which the distance between two points is determined as $d(x,y) = \sqrt[p]{|x_1 - y_1|^p + |x_2 - y_2|^p}$. Another metric with applications for cranes and drills is the Manhattan metric $d(x,y) = |x_1 - y_1| + |x_2 - y_2|$. Many results like the computation of the Voronoi diagram can be adapted to different metrics.

7 Spatial Pruning

One possibility to make the search space of Dijkstra's or the A* algorithm smaller is to prune successor states, i.e., to ignore some neighbor points in the inner loop. Here we introduce shortest path pruning based on precomputed distance information in form of a certain bounding box for each edge. A pseudo-code characterization of the modified A* search algorithm can be found in [8]. Other bounding containers are discussed in [37]. The neighbors – or more precisely the incident edges to these neighbors – that can be ignored safely are those that are not on a shortest path to the target. So the two stages for geometric speed-ups are as follows:

1. In a preprocessing step, for each edge, store the set of nodes that can be reached on a shortest path that starts with this particular edge.
2. While running Dijkstra's algorithm or A*, do not insert edges into the priority queue that are not part of a shortest path to the target.

The problem that arises is that for n nodes in the graph one would need $O(n^2)$ space to store this information, which is not feasible even for contracted graphs. Hence, we do not remember the set of nodes that can be reached on a shortest path for an edge, but approximations of it, so-called containers. The required storage will be in $O(n)$ in total, but a container may contain nodes that do not belong to this set. Note that this does not hurt an exploration algorithm in the sense that it still returns the correct result, but increases only the search space.

More formally, for $e = (u, v) \in E$ let $C(e)$ be the set of nodes t, so that there exists a shortest path from u to t that traverses e. For all $e = (u, v) \in E$ the smallest rectangular box $[x_1, x_2] \times [y_1, y_2]$ is selected, in which all t in $C(e)$ are contained; i.e.,

$$\begin{aligned} x_1 = \min_{t \in C(e)} \{t.x\},\ x_2 &= \max_{t \in C(e)} \{t.x\}, \\ y_1 = \min_{t \in C(e)} \{t.y\},\ y_2 &= \max_{t \in C(e)} \{t.y\}, \end{aligned}$$

where $t.x$ and $t.y$ are the x- and y-coordinates of t, respectively. Incorporating the above geometric pruning facilities into an exploration algorithm will retain its completeness and its optimality, since at least one shortest path from the start to the goal node will be preserved. Since it refers to the layout of nodes only, it also applies to the contracted graph that we have constructed. The pruning is based on the computation of all shortest paths that pass the edges e in E. With Johnson's algorithm [6] pre-computation is available in time $O(n^2 \log n)$.

Bounding-box pruning extends early observations of [34], where angular sectors of all shortest paths that pass an considered edge were used. More formally for all $e = (u, v)$ in E a pair of half-planes $h_l(e)$ and $h_r(e)$ is selected that is minimal enclosing for all t in $C(e)$. The half planes h_l and h_r are stored by memorizing two representative nodes. The inclusion query for a goal node t can be answered in constant time through processing trigonometric-free computations.

8 Dynamic Planning

Consider the scenario when while following a shortest path returned by the system, the user finds out that because of a road accident or some other disaster, there is a traffic jam. This results in an increase in travel time that was otherwise needed to pass through that area. In this case a second shortest path from the current position to the destination is needed that also considers the affected area.

In our case this situation is depicted as the increase in the weight of the edges in the affected path. This implies the invalidity of some of the bounding boxes, particularly the ones that contain edges with increased edge weights. This, in turn, restricts the bounding-box searching algorithm to make use of the precomputed information.

We say that a constraint is a *soft* constraint if the weight of an edge is increased only by a constant number. It represent the situations, where e.g: only 3 out of 4 lanes of a street are affected. The extreme case, when the weight is increased to $+\infty$, hence making the edge completely unusable, is characterized as a *hard* constraint.

This section presents two approaches of introducing dynamics in the system. In the first approach, due to a disaster, we assume that some particular edges are directly affected and their weights are increased. While, in the second approach we represent a disaster as a geometrical object on the graph, that affects all the edges covered by that object.

In both of these models we assume that the changes in the graph are temporal in nature, i.e., they disappear after some time. In case of persistent changes, i.e. where it is desirable to update the bounding boxes, we refer the reader to [38].

8.1 Individual Edge Dynamics

In this model, the disasters are represented by a temporal change in the weight of an edge. In order to incorporate and keep track of the changes we extend the structure of our graph to $G_c = (V_c, E_c, d_c, b, \delta, b_\delta)$, where $\delta : E \rightarrow I\!R$ is the change in the weight of an edge due to some disaster and $b_\delta : E \rightarrow Time$ is the time stamp after which a change is considered as *void*.

Due to the change in the edge weights, some of the bounding boxes in the graph become useless; precisely the ones that contain affected edges. This characterization along with the containment property between the bounding boxes of the consecutive edges, give us some hints in utilizing the bounding boxes even in a dynamic setting.

If $q = (s, t)$ be the user's query, let the bounding box of node s be defined as

$$BB(s) = \bigcup_{\forall e \in OutEdges(s)} BB(e).$$

We can now check the validity of the bounding box pruning scheme by taking the intersection of $BB(s)$ with the affected edges meanwhile considering the time of the query and b_δ for the validity of disasters.

An empty intersection depicts the situation, where the bounding box pruning scheme is still valid for that particular query. In case of a non-empty intersection the search algorithm does not use the pre-computed information and relies on the run-time heuristics like A*.

Figure 6 displays a graph with the bounding boxes, attached only to the edges that are on the shortest path from s to t, shown with the dashed lines. The shortest path is shown with a bold line. In Figure 7 we show the scenario when the weight of the edge (s, a) increases by 1 and because of this increase the old shortest path between s and t seized to exist as the shortest path. The new shortest path is the one shown with the bold-dashed line having the cost 9.82. Observe that due to this increase only the bounding box containing the edge (s, a) is affected and the rest of the bounding boxes can be used to guide the shortest path searching without any problem.

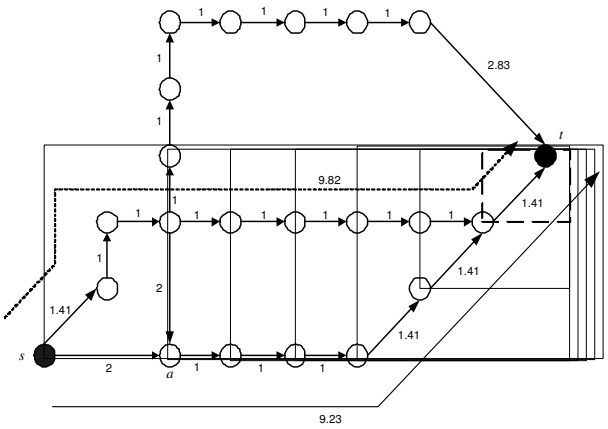

Fig. 6. A graph annotated with bounding boxes

8.2 Disasters as Geometrical Objects

In this model, disasters are represented as geometrical objects on top of the graph and affecting the area of graph underneath them. For simplicity we represent disasters as rectangles. The graph can now be extended to $G_c = (V_c, E_c, d_c, t, \delta, \Gamma, b_\Gamma)$ where Γ represent the set of disaster's geometrical objects, and $b_\Gamma : \Gamma \rightarrow Time$ gives the time stamp after which an object o is not valid anymore. Mapping δ represent the change in the weight of an edge. An object $o \in \Gamma$ consists of the position and dimensions of rectangle and the estimate that should be imposed on the edges underneath o.

We identify two approaches to proceed in this model. In the first approach we check the validity of $BB(s)$ by searching for the intersection between $BB(s)$ and Γ. This problem is studied in the domain of Computational Geometry as *Rectangle Intersection Problem* [30] and can be solved in time $O(n \log n + k)$ using $O(n \log n)$ space, where $n = |BB(s)| + |\Gamma|$ and k is the number of reported intersections. The algorithm uses a combination of range and segment trees. The space complexity can be reduced to $O(n)$ by using interval trees in place of segment trees.

In case of invalid $BB(s)$, the pre-computed information can not be used. At this point we go through Γ and update the δ values for the affected edges. We can then proceed to searching as describe earlier using $d_c + \delta$ as the weights of the edges.

The updating of δ can be automated in an off-line procedure that is triggered on the arrival of a new disaster object or the time-out of an existing one.

The second approach depicts a more real-world scenario. We utilize the fact that it is probable that while going along a path, a timeout of some objects occur hence rolling back the weights of the affected edges to their original values.

During exploration in the search algorithm, we check the collision with an object o and introduce its affects to the edges covered only if the user's query time plus the time required to the current position is less than $T_\Gamma(o)$.

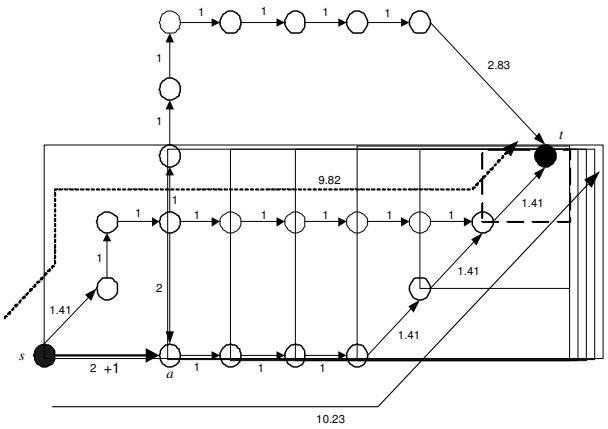

Fig. 7. An annotated graph with increased edge

9 GPS-ROUTE

We have built our generic system GPS-ROUTE[11] on top of the LEDA algorithm library [27] that supports accurate and efficient geometry and graph algorithms. To read GPS data, we wrote a GPS trace parser that generates the set of LEDA points and segments and that extends the existing structures with according time values. This allows to query combined shortest distance and time paths. GPS-ROUTE is designed to be open, to allow sharing their traces via an appropriate Internet portal, using digitized maps and posing shortest path queries.

By functionality, GPS-ROUTE can be divided into two parts: *Preprocessor* and *Shortest Path Algorithm*. The *Preprocessor* itself consists of all the classes required for trace parsing and filtering, graph construction, node localization, and graph compression. The preprocessor is designed to be easily extensible and flexible by having its foundation on interface classes, that are to be extended into specialized components such as different compression schemes or different filters. The second part, the *Shortest Path Algorithm*, consists of the algorithm to be used to answer user's query of shortest paths. It also consists of the graph annotation routines to prune the search space during searching. Figure 8 shows the design of the *Preprocessor*.

Currently, GPS-ROUTE can be accessed by two different interfaces:A web-based text-only interface and a more sophisticated graphical interface. The web-based interface can be invoked by providing the GPS traces and the query points. In the presence of a path between the query points, the path is returned in the form of points that can be uploaded to a GPS receiver for navigation. The other interface is VEGA [17], a client-server architecture that runs executables on server side to be visualized on client side, written in Java. The client is used both as the user front-end and the visualization engine and can be embedded in a web page as a JAVA Applet. VEGA allows the visualization

[11] The Internet interface is available at `ad.informatik.uni-freiburg.de/~edelkamp/gpsroute`

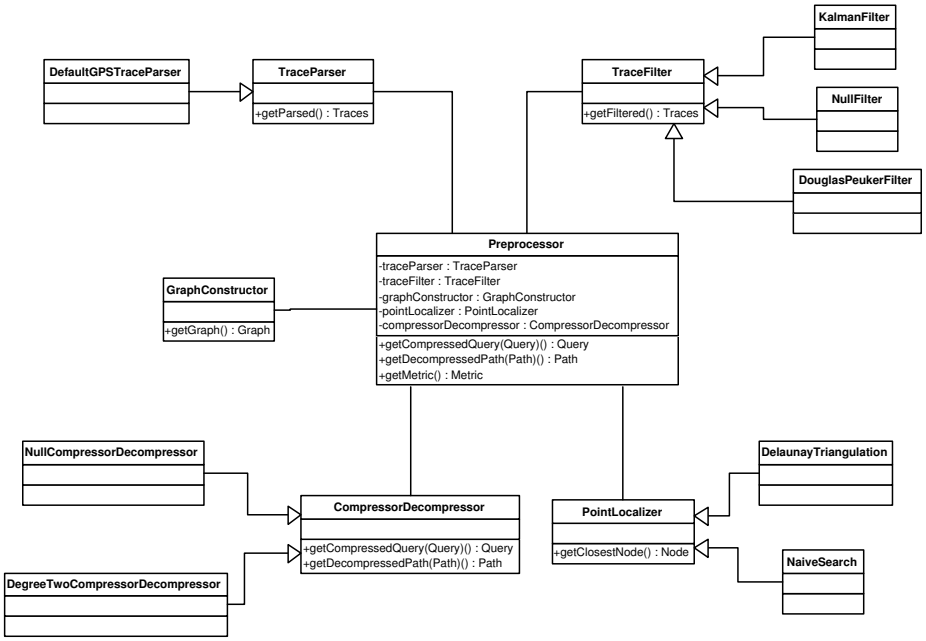

Fig. 8. Design of Preprocessor.

of different steps of the GPS-ROUTE like displaying the original graph constructed from the traces and then showing the compression graph and the searched path on top of it. The manipulation of graphs' attributes can be done in a very user-friendly way. It also allows the export of displayed graphs to XFIG format.

The generated graphs and path can be overlaid on the calibrated maps for a more real-world visualization. Calibrated maps can be downloaded from the Internet, scanned from ordinary ones, or extracted from software mapping tools, e.g. by the ones that are used and distributed by surveying authorities[12]. Figure 9 displays a topographic map[13] with the path shown as a dark-gray trace.

The VEGA server allows `C/C++` algorithms to be easily accessible over network using TCP/IP. The server maintains a list of installed algorithms and allows the client to select, provide input, and visualized the output of the algorithms.

Even though adapting VEGA for trajectory planning yields a convenient user interface with a wide range of applicability, it is written in JAVA so that large graphs and trails pose some burden in illustrating them. Therefore, reduction schemes are mandatory for real-world applications.

For a fair empirical analysis of the underlying planning algorithms, we use the fact that VEGA executables perform communication via standard I/O and can be invoked without starting the visualization at all. Our results were preliminary in the sense that

[12] `www.adv-online.de/produkte/top50`
[13] `www.geodaten.bayern.de`

Fig. 9. Topographic map with the selected route.

only sample GPS trajectory sets were considered. We chose the library LEDA, version 3.6.1, since this was one of the last one that could be used free of charge for research.

10 Experiments

This section presents the results of the experiments done with our current implementation of GPS-ROUTE. All running times are given in seconds and measured on a 248 MHZ Sun Ultra workstation. We experimented with two data sets: one gathered on a bicycle and one collected with a taxi. Both data sets were produced in a frequency of 1 Hz. Even for this small and moderately sized data sets, we can exhibit some effects of the proposed acceleration features. The experimental data matches the study in [8].

#points	#queries	t_c	t_s	t'_s
1,277	1,277	0.10	0.30	12.60
1,706	1,706	0.24	0.54	24.29
2,365	2,365	0.33	1.14	43.3
50,000	50,000	13.73	14.26	$>$ 10,000

Table 1. Effect of efficient point localization.

Table 1 compares the performance for diagram construction (t_c) and searching (t_s) query points to the naive search scheme (t'_s). We posed as many queries as there were points, by giving a small offset to the original point coordinates. Localization queries have a very small accumulated running time, showing that pre-computation is crucial.

In Table 2 we depict the running time of the sweep-line algorithm as well as the effect of heuristic search, where t_g is the time of the sweep-line algorithm, t_c is the preparation time of the search algorithm (initializing the data structures) t_s is the pure searching time for a single shortest path query, and #exp is the corresponding number of expansions done in computing the shortest path.

As in the case of point localization the sweep-line intersection algorithm is more time consuming than all further computations. With about a second CPU time, prepar-

	#points	t_g	t_c	t_s	#exp
Dijkstra	1,277	0.42	0.01	0.01	1,293
A*	1,277	0.42	0.01	0.00	243
Dijkstra	1,706	0.27	0.01	0.01	1,421
A*	1,706	0.27	0.00	0.00	451
Dijkstra	2,365	0.37	0.00	0.01	1,667
A*	2,365	0.37	0.00	0.01	1,600
Dijkstra	50,000	11.13	0.27	0.27	44,009
A*	50,000	11.13	0.26	0.20	18,755

Table 2. Effect of heuristic search.

ing and running a shortest path query is fast. In fact, initialization time of the data structures can be avoided through hashing. This proves that pre-computation for an on-line query system pays off. For heuristic search, we obtained a significant reduction in the number of expanded nodes. However, the observed CPU gain in the example is small.

#points	#nodes	#comp	t_c	#exp	t_s
1,277	1,473	199	0.01	48	0.00
1,706	1,777	74	0.02	35	0.00
2,365	2,481	130	0.03	72	0.00
50,000	54,267	4,391	0.59	1,738	0.02

Table 3. Effect of compression.

Next, all nodes of degree two were deleted by adding up distance and time values. Table 3 depicts the number of original data points, the size of the overlaid and compressed graph, the performance of compression (t_c), the number of expanded nodes in the A* algorithm and corresponding search CPU time for one shortest path query (t_s). As expected, compression drastically reduces the graph complexity, and in turn the subsequent search efforts.

In Table 4 we evaluate the effect of geometric pruning on our small test set for the compressed graphs, since the pre-computation times for the original graphs were large (50.06 and $>$ 10,000 seconds, respectively). We run the combination of Dijkstra/A* with bounding box pruning for computing shortest path on 200 successive random queries. As we see, the work for pre-computing all shortest pairs (t_c) can be large. This is counter-balanced with a significant gain in the number of expanded nodes (primed variable denote the original algorithm). For compressed graphs we observe a factor of 2-4, with better performance for larger graphs. The time gain is much smaller burdened by the number of additional comparisons and path extraction. Heuristic search can successfully be combined with geometric pruning. The smaller impact of heuristic search compared to Table 2 can be attributed to the averaging effect of random queries, posing easier exploration problems compared to the selected extreme.

	#nodes	t_c	t_s	#exp	t'_s	#exp'
Dijkstra	199	1.87	0.34	6,596	0.60	19,595
A*	199	1.87	0.26	3,135	0.19	7,912
Dijkstra	74	0.52	0.30	2896	0.29	7271
A*	74	0.52	0.28	2762	0.30	5169
Dijkstra	130	1.14	0.49	4144	0.54	12392
A*	130	1.14	0.49	3848	0.56	10060
Dijkstra	4,391	1,299	9.36	101,064	17.18	458,156
A*	4,391	1,299	8.11	65,726	12.88	217,430

Table 4. Effect of pruning.

We furthermore observed that geometric cuts perform good in two cases. First, if test data contains many paths to the target. The exploration algorithm is slow then, because it does not know which route to take, i.e. there are many possible neighbors that it has to consider. When excluding some of them, then the search space is much smaller. If there is no path to the target at all, bounding boxes also help: For all edges, the target is then not in the set of nodes that can be reached on a shortest path starting with this edge. It is therefore (maybe) not in the bounding box that belongs to this edge. In the ideal case, for a query with no solution, restricted Dijkstra only looks at the source and the incident edges.

Finally, in Table 5 we measured the time of decompression of the compressed shortest path. As we can see, in the larger graph, decompressing 200 shortest paths is almost as fast as compressing the entire graph once.

#points	#queries	t_c	t_d
199	200	0.01	0.09
74	200	0.02	0.15
130	200	0.03	0.28
4,391	200	0.59	0.65

Table 5. Effect of decompression.

11 Related Work

This paper does not address the issue of statistical clustering of GPS data to automatically infer a map by condensing the data set through road centerlines and clustering. The interested reader is referred to precursor work in [11, 33], where in the context of car navigation, lane-precise maps are inferred. The result will be more accurate, cheaper, up-to-date maps. The work provides a domain dependent system that automatically generates digital road maps that are significantly more precise and contain descriptions of lane structure, including number of lanes and their locations, and also

detailed intersection structure. Therefore, the work suggests to apply geometric algorithms instead for efficient shortest path route planning. With this respect no running system was provided. Moreover, the preliminary treatment lacked advanced filtering and reduction techniques, efficient nearest neighbor calculations for the queries and acceleration techniques.

Due to the intense industrial interest, GPS[14] navigation is a rising research field with several publications to certain application areas and current technology. Even first conferences such as ION GPS-2002 in Portland, Oregon[15] have been initiated. At Stanford university, the GPS laboratory [16] consists of a group of 30 researchers. Domain-independent geometric and integrated approaches similar to the one presented here, however, are very difficult to obtain. The closest match we found is the work of the machine learning research lab at DaimlerChrysler, Palo Alto, with a branch that concentrates on DGPS map generation for cars.

Computing the travel graph according to a set of splines is also possible, but calls for refined algorithmic issue as addressed in the EXACUS project[17] at MPI, Saarbrücken. Our current implementation restricts to line representations.

12 Conclusion

Heuristic search is almost universal in accelerating solution path search and has be combined with several algorithmic extensions to avoid state space explosion: iterative deepening [21], transposition tables [32], bidirectional search [22], finite state pruning [35], frontier search [24], bit-state hashing [18], binary decision diagrams [9] partial order reduction [26], symmetry reduction [25], just to name a few.

In difference to the above approaches in this paper we exploit the spatial structure of the problem domain to exclude successor states from consideration. Even though the explicit graph might be too large to be kept in main memory, exploration through the compressed one might still be possible. Similar to pattern databases [23], we generated a database of shortest path information before the query is processed. In difference to FSM pruning the approach does not rely on the regular structure of the domain, but on a graph embedding in the plane.

Our search engine is designed to answer distributed shortest path queries in a very short time by preprocessing the internal information and by exhibiting the Euclidean layout of the superimposed trace graph. The pre-computation time is large, but in an on-line scenario this might be acceptable.

In future work we aim at a new option of pruning state spaces for optimal action planning. With each operator we associate pre-computed information on the set of states that are reachable on a shortest path including this operator. Since the set of states may become too large with respect to main memory capacity, it can be reduced through available boolean over-approximations.

[14] cf. link list at `www.cla.sc.edu/geog/rslab/gps.html`
[15] `www.ion.org/meetings/gps2002program.cfm`
[16] cf. `www.stanford.edu/group/GPS`
[17] `www.mpi-sb.mpg.de/projects/EXACUS`

The edge pruning rules are derived prior to the search based on a forward traversal in state space starting with the preconditions of each operator. They are concisely represented in form of binary decision diagrams. Through existential quantification of variables, an on-the-fly over-approximation scheme seems available to reduce the representation size in successive image computations.

Acknowledgment Thanks to Lioudmila Belenkaia for links to trace data reduction and to Jaspar Möller for his student project report on geometric cuts. The work is supported by DFG in the projects *heuristic search* and *directed model checking*.

References

1. I. J. Balaban. An optimal algorithm for finding segment intersection. In *ACM Symposium on Computational Geometry*, pages 339–364, 1995.
2. J. L. Bentley and T. A. Ottmann. Algorithms for reporting and counting geometric intersections. *Transactions on Computing*, 28:643–647, 1979.
3. B. Chazelle. Reporting and counting segment intersections. *Computing System Science*, 32:200–212, 1986.
4. B. Chazelle and H. Edelsbrunner. An optimal algorithm for intersecting lines in the plane. *Journal of the ACM*, 39:1–54, 1992.
5. F. Chin, J. Snoeyink, and C.-A. Wang. Finding the medial axis of a simple polygon in linear time. In *International Symposium Algorithms Computation (ISAAC)*, Lecture Notes in Computer Science, pages 382–391. Springer, 1995.
6. T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. The MIT Press, 1990.
7. G. L. Dirichlet. Ueber die Reduktion der positiven quadratischen Formen mit drei unbestimmten ganzen Zahlen. *Journal Reine und Angewandte Mathematik*, 40:209–227, 1850.
8. S. Edelkamp, S. Jabbar, and T. Willhalm. Geometric travel planning. In *IEEE International Conference on Intelligent Transportation Systems (ITCS)*, 2003. To appear.
9. S. Edelkamp and F. Reffel. OBDDs in heuristic search. In *German Conference on Artificial Intelligence (KI)*, Lecture Notes in Computer Science, pages 81–92. Springer, 1998.
10. S. Edelkamp and S. Schrödl. Localizing A*. In *National Conference on Artificial Intelligence (AAAI)*, pages 885–890, 2000.
11. S. Edelkamp and S. Schrödl. Route planning and map inference with global positioning traces. In *Essays dedicated to Thomas Ottmann*, Lecture Notes in Computer Science, pages 128–151. Springer, 2003.
12. S. J. Fortune. A sweepline algorithm for Voronoi diagrams. *Algorithmica*, pages 153–174, 1987.
13. L. J. Guibas, D. E. Knuth, and M. Sharir. Randomized incremental construction of Delaunay and Voronoi diagrams. *Algorithmica*, 7:381–413, 1992.
14. L. J. Guibas and J. Stolfi. Ruler, compass and computer: the design and analysis of geometric algorithms. *Theoretical Foundations of Computer Graphics*, 40:111–165, 1988.
15. S. Gutmann. Markov-Kalman localization for mobile robots. In *International Conference on Pattern Recognition (ICPR)*, 2002.
16. J. Hershberger and J. Snoeyink. An $O(n \log n)$ implementation of the Douglas-Peuker algorithm for line simplification. *ACM Computational Geometry*, pages 383–384, 1994.
17. C. A. Hipke and S. Schuierer. Vega—a user-centered approach to the distributed visualization of geometric algorithms. In *Computer Graphics, Visualization and Interactive Digital Media (WSCG)*, pages 110–117, 1998.

18. F. Hüffner, S. Edelkamp, H. Fernau, and R. Niedermeier. Finding optimal solutions to Atomix. In *German Conference on Artificial Intelligence (KI)*, Lecture Notes in Computer Science. Springer, 2001.
19. H. Kautz and B. Selman. Pushing the envelope: Planning propositional logic, and stochastic search. In *National Conference on Artificial Intelligence (AAAI)*, pages 1194–1201, 1996.
20. D. G. Kirkpatrick. Optimal search in planar subdivisions. *SIAM Journal of Computing*, 12(1):28–35, 1983.
21. R. E. Korf. Depth-first iterative-deepening: An optimal admissible tree search. *Artificial Intelligence*, 27(1):97–109, 1985.
22. R. E. Korf. Divide-and-conquer bidirectional search: First results. In *International Joint Conferences on Artificial Intelligence (IJCAI)*, pages 1184–1191, 1999.
23. R. E. Korf and A. Felner. Disjoint pattern database heuristics. *Artificial Intelligence*, 2001.
24. R. E. Korf and W. Zhang. Divide-and-conquer frontier search applied to optimal sequence allignment. In *National Conference on Artificial Intelligence (AAAI)*, pages 910–916, 2000.
25. A. Lluch-Lafuente. Symmetry reduction in directed model checking. Technical report, Institute of Computer Science at Freiburg University, 2003.
26. A. Lluch-Lafuente, S. Leue, and S. Edelkamp. Partial order reduction in directed model checking. In *Workshop on Model Checking Software (SPIN)*, Lecture Notes in Computer Science, pages 112–127. Springer, 2002.
27. K. Mehlhorn and S. Näher. *The LEDA Platform of Combinatorial and Geometric Computing*. Cambridge University Press, 1999.
28. K. Mulmuley. A fast planar partition algorithm. *Journal of Symbolic Computation*, (10):253–280, 1990.
29. J. Pearl. *Heuristics*. Addison-Wesley, 1985.
30. F. P. Preparata and M. I. Shamos. *Computational Geometry: An Introduction*. Springer, 1985.
31. D. Ratner and M. K. Warmuth. The (n^2-1)-puzzle and related relocation problems. *Journal of Symbolic Computation*, 10(2):111–137, 1990.
32. A. Reinefeld and T. A. Marsland. Enhanced iterative-deepening search. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 16(7):701–710, 1994.
33. S. Schrödl, S. Rogers, and C. Wilson. Map refinement from GPS traces. Technical Report RTC 6/2000, DaimlerChrysler Research and Technology North America, Palo Alto, CA, 2000.
34. F. Schulz, D. Wagner, and K. Weihe. Dijkstra's algorithm on-line: An empirical case study from public railroad transport. *Journal of Experimental Algorithmics*, 5(12):110–114, 2000.
35. L. A. Taylor and R. E. Korf. Pruning duplicate nodes in depth-first search. In *National Conference on Artificial Intelligence (AAAI)*, pages 756–761, 1993.
36. G. M. Voronoi. Novelle application des parametres continus a la theorie des formes quadratiques. *Reine und Angewandte Mathematik*, 133:97–178, 1907.
37. D. Wagner and T. Willhalm. Geometric speed-up techniques for finding shortest paths in large sparse graphs. In *Proc. 11th European Symposium on Algorithms (ESA 2003)*, LNCS. Springer, 2003. To appear.
38. D. Wagner, T. Willhalm, and C. Zaroliagis. Dynamic shortest path containers. In A. Marchetti-Spaccamela, editor, *Proc. Algorithmic MeThods and Models for Optimization of RailwayS 2003*, Electronic Notes in Theoretical Computer Science, 2003. To appear.
39. R. Zhou and E. Hansen. Sparse-memory graph search. In *International Joint Conference on Artificial Intelligence (IJCAI)*, 2003.