Geometric Travel Planning

Stefan Edelkamp
Universität Dortmund
Fachbereich Informatik, Lehrstuhl V
Baroperstraße 301
D-44227 Dortmund
stefan.edelkamp@cs.uni-dortmund.de
Shahid Jabbar
Universität Dortmund
Fachbereich Informatik, Lehrstuhl V
Baroperstraße 301
D-44227 Dortmund
shahid.jabbar@cs.uni-dortmund.de
Thomas Willhalm
Universität Karlsruhe
Institut für Logik, Komplexität
und Deduktionssysteme
D-76128 Karlsruhe
willhalm@ira.uni-karlsruhe.de

***Abstract*— This paper provides a novel approach for optimal route planning making efficient use of the underlying geometrical structure. It combines classical AI exploration with computational geometry.**

Given a set of global positioning system (GPS) trajectories, the input is refined by geometric filtering and rounding algorithms. For constructing the graph and the according point localization structure, fast scan-line and divide-and-conquer algorithms are applied.

For speeding up the optimal on-line search algorithms, the geometrical structure of the inferred weighted graph is exploited in two ways. The graph is compressed while retaining the original information for unfolding resulting shortest paths. It is then annotated by lower bounds and refined topographic information; for example by the bounding boxes of all shortest paths that start with a given edge.

Traffic disturbances can result in an increase in travel time for the affected area, which in turn, can affect the pre-computed information. This paper discusses two models of introducing dynamics in a navigation system.

The on-line planning system GPS-ROUTE implements the above techniques and provides a client-server web interface to answer series of shortest-path or shortest-time queries.

I. Introduction

Improved navigation is an ubiquitous need to satisfy nowadays mobility requirements. With the industrial emergence of low-cost positioning systems and by the accelerated development of hand-held devices and mobile telephones, integrated data gathering and processing to assist personal navigation becomes feasible at a very large scale.

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. Most available maps provide no information for bikers and pedestrians. In some regions of the world, digital maps are not available at all. Maps contain information on road classes and travel distances only, which is often not sufficient to infer travel time. Moreover, it is not possible to pose timed-queries to those maps i.e., to request a shortest path from a source to a destination during a specific period of time or day. The need for timed-queries arises from the observation that the travel time can vary drastically during different types of days: workdays and holidays.

The algorithmic issues to pre-process and answer queries for short and timely paths with respect to the current position have not been settled yet. Consequently, in this text we present the design and implementation of a flexible on-line information system that features different enhancements to the route planning problem based on GPS data. It is application domain independent, since all pre-processing and path planning algorithms merely refer to GPS trajectories. The off-line input is a set of traces and the on-line input is a set of path queries. These routes can be visualized on top of a topographic map or transferred to an end-user GPS device for route tracking. The incoming (possibly differential) GPS data is refined by filters that take additional inertial input sources into account.

We refer to the portfolio of algorithms as *geometric*, since all computations from trace manipulation to plan visualization exploit the underlying (Euclidean) geometry. For example we utilize the layout of the graph to speed-up shortest path search. As the graph is large, one cannot afford to use more than linear space. In a preprocessing step, for each edge geometric objects

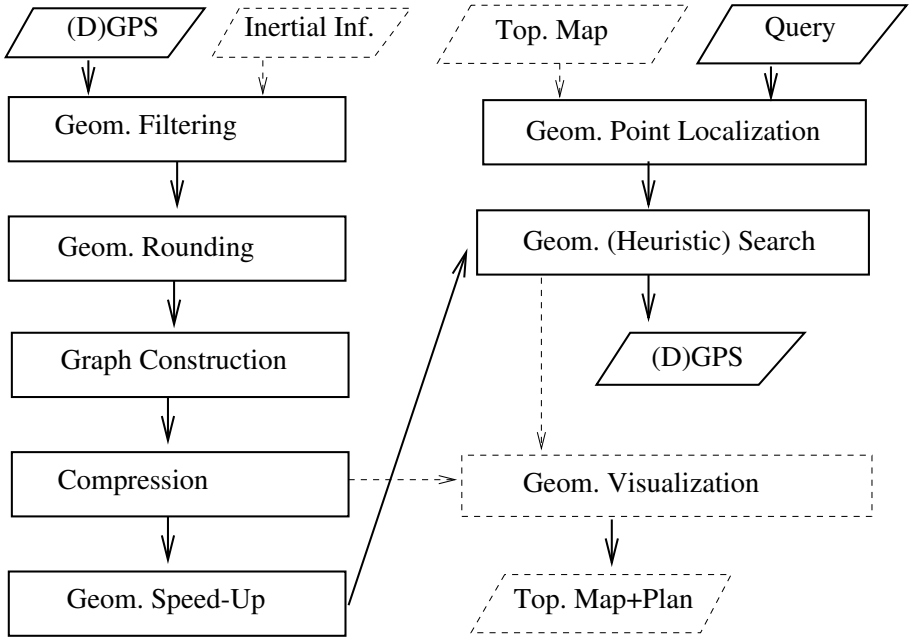

Fig. 1. The architecture of the GPS route-planning system.

are determined representing all nodes to which a shortest path using this edge exists. The object representation needs constant space per edge.

Figure 1 depicts the flow of data in our approach. Raw GPS data is fed into the system, filtered and refined. Then the trace graph is built and the compressed graph is computed on top of it. Geometric speed-up information is computed to produce the annotated graph. The route planning engine takes this extended layered graph and the provided query to compute a plan, which can be uploaded back to a GPS device for navigation. In case the provided query points do not correspond to graph nodes, we suggest to use the nearest neighboring nodes instead. The visualization step applying vertex detection, data reduction, spline fitting, etc. is optional.

This architecture also structures the rest of the paper: First we address the data refinement and geometric rounding. Then we turn to the graph construction process by applying efficient geometric algorithms to the set of GPS traces. The graph is further annotated to accelerate shortest path queries in form of Euclidean distance heuristics possibly aided by geometric shortest path pruning information. All route-finding algorithms preserve optimal routes. In the experimental section we evaluate our implementation on a few sets of collected GPS data. An overview of models for navigation in a dynamic environment is presented next. Related, current and future work is presented in Section XII followed by the conclusion.

II. Data Gathering

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 2, you see a moderately equipped bike to perform advanced mobile data gathering. At the handlebars we installed a GPS receiver[1] and an advance cycling computer[2]. 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[3] that was developed to memorize flight data. The memorized traces are transfered to a PC by a simple terminal program[4]. 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[5] that memorizes ticks each second. Besides the recording of current velocity through incoming wheel-turn ticks, a cycling computer can trace altimeter and a heart pulse data. 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. 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, including but not limited to drivers, bikers, hikers, and robots. In near future we expect all different devices to be merged into one.

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

Fig. 2. A bike equipped to gather GPS data.

III. Geometric Filtering and Rounding

Geometric filters are used to detect outliers in the GPS data. Specifying a mobility model for the moving object further helps to eliminate false signals. Inertial information like altitude, distance and angle can best be included in the actual data through the process of Kalman filtering [1], maintaining two statistical moments: the process state and the error covariance matrix. Suppose that the GPS position $l = (x, y, \theta)$ with orientation θ is updated by distance and angular change vector $a = (\delta, \alpha)$ to $F(l, a) = (x + \delta\cos\theta, y + \delta\sin\theta, \theta + \alpha)^T$. The change may lead to errors in the assumed position. Kalman filtering now assumes that distance and rotation satisfies a Gaussian distribution, i.e. $l \sim N(\mu_l, \Sigma_l)$ and $a \sim N(\mu_a, \Sigma_a)$ with means $\mu_l = (\bar{x}, \bar{y}, \bar{\theta})$ and $\mu_a = (\bar{\delta}, \bar{\alpha})$ and co-variance matrices Σ_l and Σ_a. It updates the values μ_l by $F(\mu_l, \mu_a)$ and Σ_l by $\nabla F \cdot \Sigma \cdot \nabla F^T$, where ∇F is the derivative of F and Σ the combined covariance matrix of Σ_l and Σ_a.

For further data reduction, we apply the Douglas-Peuker *geometric rounding*[6] algorithm [2]. The method was developed to reduce the number of points to represent a digitized curve from maps and photographs. It considers a simple trace[7] 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. It has latter been improved to $O(n \log^* n)$, where $\log^* n = \min\{k | \underbrace{\log\log\cdots\log}_{k \text{ times}} n = 1\}$.

[6] www.mpi-sb.mpg.de/∼mehlhorn/SelectedTopics02/GeometricRounding/GeometricRounding.html

[7] The original algorithm [3] can handle certain forms of self-intersections.

IV. Graph Construction and Compression

The travel graph is the embedded overlaid set of traces together with the according intersections. To compute the superimposed graph, the sweep-line segment intersection algorithm of [4] has been adapted. 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 comprises two data structures. In the Event Queue the active points are maintained, ordered with respect to their first coordinate. In the Status Structure, the set of active segments with respect to the sweep line is stored. At each intersection the ordering of segments in the status structure changes. After new neighboring segments are found, their intersections are computed and inserted into the Event Queue. Using a standard heap for the Event Queue and a balanced tree for the Status Structure yields an $O((n + k) \log n)$ time algorithm, with n being the number of data points and k being the number of intersections. A step towards improvement of time performance to $O(n \log^2 n / \log\log n + k)$ has been proposed by [5]. The first $O(n \log n + k)$ algorithm due to [6] used $O(n + k)$ storage. The $O(n \log n + k)$ algorithm with $O(n)$ space is due to [7].

In typical travel networks, the number of edges are proportional to the number of nodes, because the node degree is bounded by a small constant. If one can assert that the graph is planar – as in our case – the number of edges is linear in the number of nodes by Euler's formula.

Once the travel graph is built, many nodes of degree two remain. For shortest path computations these nodes can be eliminated by merging the adjacent edges through adding their distance and travel time values. Actually, only start, end

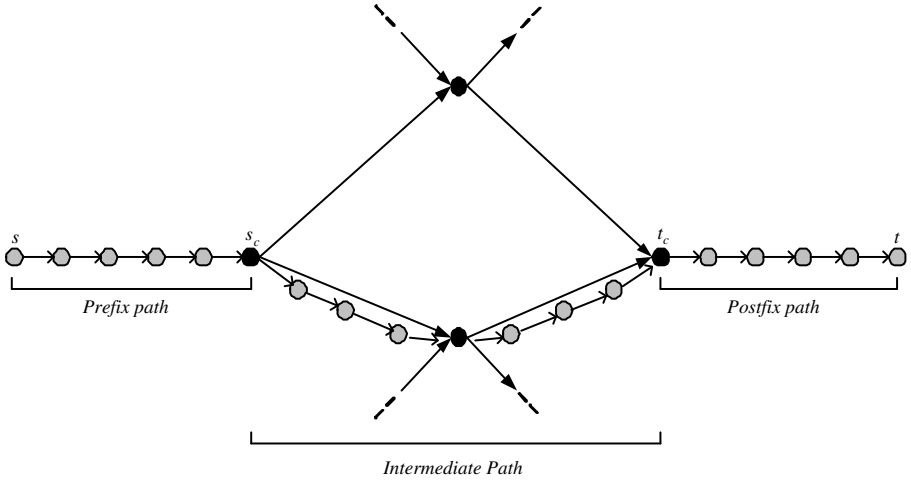

Fig. 3. Decompression of an established path.

and segment intersections points remain, reducing the space complexity of the graph from $O(n+k)$ to $O(l+k)$, where l is the number of traces and $l \ll n$. Given the original graph, the compressed graph is computed in time $O(n+k)$. The graph may loose its physical layout which for different reasons is important to retain. First of all, it would only be possible to start and end a trip on an existing compressed graph node. Moreover, to display the established route to the user and for GPS-guidance, it has to be re-embedded into the original context. Therefore, our solution is to maintain a layered graph $G = G_b \cup G_c$, where the nodes V_b in the bottom level correspond to the original (filtered) GPS data and the nodes in the top level span the compressed graph G_c.

More precisely, a compression is surjective mapping $\phi : V_b \rightarrow V_c$, so that $e_c = (u_c, v_c) \in E_c$ if there exists a path $p = u_b, x_1, \ldots, x_k, v_b$ and $indeg(x_i) = outdeg(x_i) = 1$ with $\phi(u_b) = u_c$ and $\phi(v_b) = v_c$. In practice, ϕ is computed through a linear time algorithm that goes through 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, they are deleted, else they are made hidden in order to be restored later.

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

To extract the established route for user guidance, it has to be re-embedded into the original context. Figure 3 depicts an example for decompression, where (s, t) is the query in the decompressed and (s_c, t_c) is the query in the decompressed graph. The *prefix* and *suffix* of the solution path are established by following each located start and goal node forward resp. backward in the original graph until the first compressed node is reached. These are the query nodes for the compressed graph. Since there is only one path no ambiguities can arise. If $u_1, \ldots, u_k$ was the the solution path in the contracted graph then *prefix*(s, u_1), *decompress*(u_1, u_2), ..., *decompress*(u_{k-1}, u_k), *suffix*(u_k, t) would be the according (s, t) shortest path in the original graph.

V. Node Localization

Before a query on the trace graph based on given start and goal locations can be processed, their corresponding entry nodes have to be found. 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 [8], 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. There are two main algorithms that meet the given running time: the sweep- (or beach-) line algorithm [9] and the divide-and-conquer strategy of separating paths [10]. Building a query structure on top of the Voronoi diagram is available in $O(\log n)$ time by hierarchical subdivision [11].

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 [12]. We use a sweep-line algorithm to compute a initial triangulation which is improved to a Delaunay triangulation by flipping illegal edges. The construction time is $O(n^2)$ worst case, but $O(n \log n)$ with high probability. In point localization we temporarily insert nodes into the Delaunay diagram, so that the nearest neighbor is found on an adjacent edge.

VI. Search Models

In the basic model the environment is modeled by a graph $G_c = (V_c, E_c, w_c, T_c)$ where V_c is the set of vertices and E_c is the set of edges. The function $w_c : E_c \rightarrow \Re^+$ assigns weight information with every edge. The function $T_c : E_c \rightarrow Time \times Time$ gives the temporal information about an edge,

precisely - the start time and the end time when the edge was traveled along. For notational simplicity, let us assume that T_c itself consists of two sub-functions: $T_{start} : E \rightarrow Time$ and $T_{end} : E \rightarrow Time$ that when given an edge return the start time and the end time of that edge, respectively. A query to the system is a pair of nodes (s, t). It aims at the shortest path with respect to weights w_c.

The basic model can be extended by allowing the graph to be queried not only for the shortest path but also for the $quickest$ path from s to t or for a weighted combination of both distance and time. The query $q = (s, t, \tau)$ can be posed to the system where, $\tau \in [0, 1]$ is the *preference* parameter and is used to give the preference of travel time over the distance. Based on τ, the new weight w_{new} of an edge e can be calculated as

$$w_{new}(e) = \tau \cdot [T_{end}(e) - T_{start}(e)] + (1 - \tau) \cdot w_c(e)$$

The shortest path algorithm returns the shortest path, if one exists, based on new weight information w_{new}.

It's a general observation that the travel time on a specific path can vary based on the time of the day - the travel time during rush hours is not same as during the night. Several traffic models have been proposed that model this variation. Using this observation we now allow our graph to be queried for the quickest path to be traveled at a particular time. Formally, a query is a quadruple, $(s, t, time_s, \epsilon)$, where $time_s$ is the desired start travel time. It is posed to the system for a quickest path that is *also* feasible with respect to time at every edge in the path up to ϵ time unit. We say that an edge $e = (u, v)$ is *feasible* if the difference between the minimum time to reach u and the starting time stamp at e is bounded from above and below by ϵ.

In the search algorithm, feasibility can be insured by checking for the following condition before the expansion of a node u along the edge e: $|time_s + t_{min}[u] - T_{start}(e)| \leq \epsilon$, where $t_{min}[u]$ is the minimum time required to reach u from s.

VII. Geometric Graph Search

Searching for one single-pair shortest route in the inferred lower level travel graph can sufficiently good be achieved with a single run of the algorithm of Dijkstra. For $n' = n+k$ nodes, the choice of Fibonacci heaps yields an $O(n' \log n')$ algorithm, which, given a small value of k, is about as efficient as graph construction. Moreover, the search is terminated, once the goal node has been found. Note that prior to the search, the nearest trace nodes of start and goal can be found with a scan through the data. Fixing only the compressed graph structure, the graph search complexity decreases to $O((l+k) \log (l+k))$. However, one-shot queries are not realistic. Modern navigation systems provide their services through Internet portals, so that portable devices access large databases through communication with a running server. Therefore, we assume that the set of GPS trace data is kept, updated and queried in a large-scale server system, which is required to answer many shortest path or time queries in a very short time. In the following we address efficient algorithms and data structures to reply frequent online queries. Most of the algorithms exhibit the fact that the graph is embedded in the Euclidean plane, so that refined geometric information on the set of all possible shortest paths can be associated to nodes or edges.

A. Geometric Heuristic Search

Heuristic search [13] is a well-known technique to reduce the number of expansions for a shortest path query in an implicitly given graph. This technique of goal direction includes an additional node evaluation function h into the search. For the goal node t we have $h(t) = 0$. The lower bound estimate h, also called admissible heuristic, approximates the shortest path distance from the current node to one of the goal nodes. A heuristic is *consistent*, if $w(u, v) + h(v) - h(u) \geq 0$ for all $(u, v) \in E$, where w is the weight function in the graph.

Consistent estimates are admissible. Let $p = (v_0, \ldots, v_k)$ be any path from $u = v_0$ to $t = v_k$. Then we have $w(p) = \sum_{i=0}^{k-1} w(v_i, v_{i+1}) \geq \sum_{i=0}^{k-1} (h(v_i) - h(v_{i+1})) = h(u) - h(t) = h(u)$. This is especially true if p is an optimal path from u to t.

A* with admissible estimates may result in the reopening of the nodes already visited and deleted from the priority queue. If the heuristic is consistent, the combined merit of generating path length and estimate monotonic increasing during exploration, so that no reopening can occur [13].

The lower bound that is applied to accelerate route planning is the Euclidean distance, i.e., $h(u) = ||t - u||_2$, which measures the *flight distance* to the goal node. The heuristic is consistent by the triangle inequality in Euclidean space, that is $||u - v||_2 + ||v - t||_2 \geq ||u - t||_2$ for all nodes u, v.

In the following, we will prove that in the compressed graph G_c, the new edge costs result in a consistent heuristic.

Theorem (Consistency of h) In the compressed graph G_c, h is consistent, i.e., for all $(u, v) \in V_c$ we have $w_c(u, v) + h(v) - h(u) \geq 0$.

Proof: Let us assume that the condition does not hold i.e., $w_c(u, v) + h(v) - h(u) < 0$ or $w_c(u, v) < h(u) - h(v)$. From to the definition of h, $h(u) = \|u - t\|_2$, we can rewrite the above inequality as:

$$w_c(u, v) \quad < \quad \|u - t\|_2 - \|v - t\|_2, \tag{1}$$

since a straight line distance is a lower bound on the distance value w_c of an edge. We can safely replace w_c by $\|u - v\|_2$ in Inequality 1 and obtain the following:

$$\begin{aligned} \|u - v\|_2 &< \|u - t\|_2 - \|v - t\|_2 \\ \|u - v\|_2 + \|v - t\|_2 &< \|u - t\|_2, \end{aligned}$$

in contradiction to the triangular property in Euclidean space. ■

A* for consistent estimates with respect to Dijkstra's algorithm simply changes an edge weight $w(u, v)$ to $w(u, v) + h(v) - h(u)$ given an initial offset $h(s)$. On every path from the initial state to a goal node the accumulated heuristic values telescope, and since goal nodes have estimate 0, in both algorithms the priority values at termination time are the same. Hence, at least for consistent estimates, A* (without any re-opening strategy) is complete and optimal.

If one does not search the shortest path in terms of travel distance, but the shortest path in terms of time, the *quickest path*, the lower bound has to be modified. This can be done by dividing it by an upper bound for speed.

If the graph is queried for the quickest path, we observe that by a simple extension to the model and by assuming a maximum speed μ of the mobile object, a heuristic estimate can be given for the time required to travel a path between two nodes. The heuristic estimate h' for a node u can be defined as:

$$h'(u) = \frac{1}{\mu} \cdot \|u - t\|_2$$

Admissibility of h' follows from choosing the most optimistic option both in terms of speed and distance.

If a weighted combination of both the quickest and shortest paths is required, a heuristic estimate consisting of a linear combination of both h and h' can be used.

We define h'' as the heuristic estimate for a node u in the time model as:

$$\begin{aligned} h''(u) &= \tau \cdot \frac{1}{\mu} \cdot \|u - t\|_2 + (1 - \tau) \cdot \|u - t\|_2 \\ \text{or, } h''(u) &= \|u - t\|_2(\tau \cdot \frac{1}{\mu} + (1 - \tau)) \end{aligned} \quad (2)$$

Since τ and μ are constant for the entire graph and $\|u - t\|_2$ never overestimates the actual edge cost, h'' never overestimates the actual path cost - implying h'' as an admissible heuristic.

As with the consistency of h, we can prove the consistency of h'' by proving that the edge weights remain non-negative during exploration. Recall that the weight of an edge e in the extended model is $\tau \cdot T_c(e) + (1 - \tau) \cdot w_c(u, v)$.

Theorem (Consistency of h'') In the compressed graph G_c, h'' is consistent, i.e., for all $(u, v) \in V_c$ we have $(\tau \cdot T_c(e) + (1 - \tau) \cdot w_c(u, v)) + h''(v) - h''(u) \geq 0$.

Proof: Assume that the condition does not hold, i.e., $(\tau \cdot T_c(e) + (1 - \tau) \cdot w_c(u, v)) + h''(v) - h''(u) < 0$ or $\tau \cdot T_c(e) + (1 - \tau) \cdot w_c(u, v) < h''(u) - h''(v)$ According to Equation 2, $h''(u) = \|u - t\|_2(\tau \cdot \frac{1}{\mu} + (1 - \tau))$. We can rewrite the above inequality as $\tau \cdot T_c(e) + (1 - \tau) \cdot w_c(u, v) < \|u - t\|_2(\tau \cdot \frac{1}{\mu} + (1 - \tau)) - \|v - t\|_2(\tau \cdot \frac{1}{\mu} + (1 - \tau))$.

Due to compression of straight line distances we know that $w_c(u, v) \geq \|u - v\|_2$. Also, since the time estimate to traverse an edge with maximum speed never overestimates the actual time, we can say that $T_c(e) \geq \frac{1}{\mu}\|u - v\|_2$. The above inequality can be rewritten as: $\tau \cdot \frac{1}{\mu}\|u - v\|_2 + (1 - \tau) \cdot \|u - v\|_2 < \|u - t\|_2(\tau \cdot \frac{1}{\mu} + (1 - \tau)) - \|v - t\|_2(\tau \cdot \frac{1}{\mu} + (1 - \tau))$. Hence $\|u - v\|_2(\tau \cdot \frac{1}{\mu} + (1 - \tau)) < \|u - t\|_2(\tau \cdot \frac{1}{\mu} + (1 - \tau)) - \|v - t\|_2(\tau \cdot \frac{1}{\mu} + (1 - \tau))$ and $\|u - v\|_2 < \|u - t\|_2 - \|v - t\|_2$, so that $\|u - v\|_2 + \|v - t\|_2 < \|u - t\|_2$, in contradiction to the triangular property in Euclidean space. ■

B. Geometric Pruning

Another possibility to make the search space of Dijkstra's or the A* algorithm smaller is to ignore some neighbor points in the inner loop. 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 the bounding box in the layout of the graph. The required storage will be in $O(n)$ in total, but a bounding box 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. Incorporating the above geometric pruning facilities into an exploration algorithm like Dijkstra or A* 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.

A pseudo-code implementation is given in Figure 4. For sake of clarity, we have omitted source fragments to memorize expanded nodes and to prevent the algorithm of reopening.

The pruning is based on the computation of all shortest paths that pass the edges e in E. Using Fibonacci heaps, for all edges this gives us an amortized worst case pre-compilation time of $O(n^2 \log n)$ in total, where n is the number of nodes in the graph. Since the original graph is considerable large, we apply the algorithm only for the contracted graphs. However, in difference to the $O(n^3)$ all-pair shortest path algorithm of Floyd and Warshall [14], the space requirements are linear, since no adjacency matrix representation is needed.

A* with Geometric Speed-Up:

```
Priority Queue Q ← {(s, h(s))}
while (Q ≠ ∅)
  u ← DeleteMin(Q)
  if u = t return u
  for all neighbors v of u
    if t is inside BBox[(u, v)]
      f'(v) ← f(u) + w(u, v) + h(v) − h(u)
      if (Search(Q, v))
        if (f'(v) < f(v))
          DecreaseKey(Q(v, f'(v))
      else Insert(Q, (v, f'(v))
```

Fig. 4. Implementation of bounding box pruning in A*.

C. Processing Negative Edge Weights

Even in our application we can think of edges that are assigned to a negative value, e.g. when ranking the benefits

Fig. 5. Topographic map and selected route.

of traversing edges. Johnson's algorithm [14] shows that pre-computation in time $O(n^2 \log n)$ can also enable us to handle the case of negative edge weights. In some sense, the approach is opposed to heuristic search, since a negatively weighted graph is transfered to a positively weighted one, in advance to the main computations. Here the re-weighting function h is obtained through an initial run of Bellman-Ford's algorithm, that is $h(u)$ is the shortest path value from a fixed source s to u. If the graph contains negative-weight cycles, Bellman-Ford will detect that. Since storing h consumes linear space on-line queries based on geometric cuts can be made available even in negatively weighted graphs through h.

VIII. GPS-ROUTE

We have built our generic system GPS-ROUTE[8] on top of the LEDA algorithm library [15] 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.

The ASCII web-interface simply reads the GPS queries in form of GPS data point and displays the established route. A second user interface is base on VEGA [16], 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. VEGA allows different server and algorithm selections, input of data, running and stopping algorithms, and customization of the visualization. It can be used to display route information and to apply algorithms to selected data in a view, to control the plan execution using a VCR-like panel or the execution browser, to adjust view attributes directly or using the object browser, to show several algorithms simultaneously in multiple scenes and open different views for a single scene, to load and to save single scenes, complete runs, and attribute lists, and to export scenes in XFIG or GIF format.

The main purpose of the server is to make `C/C++` algorithms accessible through TCP/IP. The server is able to receive commands from multiple clients at the same time. It allows the client to choose from the list of available algorithms, to retrieve information about the algorithm, to specify input data, to start it and to receive output data. The server maintains a list of installed algorithms. This list may be changed without the need of stopping and restarting the server.

Calibrated maps can be down-loaded 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[9]. Figure 5 displays some topographic map[10].

Traces (and maps) have to be provided before the route planning algorithm, on which on-line queries are executed. In Figure 6 we depict the result of one shortest path query on a sample uncompressed graph structure. The highlighted path is in fact the shortest possible, since the underlying graph generated by the trail is directed. It further switches from one partial trail to another through an intersection point.

IX. Experiments

In the experiments we restrict to the basic search model. Our results were preliminary in the sense that 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. All running times are given in seconds and measured on a 248 MHz Sun Ultra workstation. We experimented with four data sets: two were gathered on a bicycle, one as a pedestrian, and one collected with a taxi. Even for this small and moderately sized data sets, we can exhibit some effects of the proposed acceleration features.

#points	$\theta = 10^{-7}$	10^{-6}	10^{-5}	10^{-4}	10^{-3}
1,277	766	558	243	77	22
1,706	1540	1162	433	117	25
2,365	2083	1394	376	28	7
50,000	48,432	42,218	17,853	4,385	1,185

TABLE I
Reduction of traces with Douglas-Peuker.

In Table I we show the effect of geometric rounding by Douglas-Peuker with different threshold values on GPS traces.

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

[9]`www.adv-online.de/produkte/top50`
[10]`www.geodaten.bayern.de`

Fig. 6. Result of a shortest path query for a small bicycle trace.

The accuracy θ is drawn on top of the table and is measured with respect to raw GPS latitude and longitude format. The running times for all executions were within one CPU second. The obtained reduction ratio for car traveling traces is smaller than for the bicycle data set. This is due to the fact that the speed of cars makes GPS data points sparse.

#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 II

EFFECT OF EFFICIENT POINT LOCALIZATION.

Table II compares the performance for Delaunay 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 III 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, preparing 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	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 III

EFFECT OF HEURISTIC SEARCH.

#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 IV

EFFECT OF COMPRESSION.

Next, all nodes of degree two were deleted by adding up distance and time values. Table IV 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.

	#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 V
EFFECT OF GEOMETRIC PRUNING (ON-LINE).

We evaluate the effect of geometric pruning in on-line setting on the compressed graphs with the compression of graphs and calculation of bounding-boxes done off-line. We run the combination of Dijkstra/A* with bounding box pruning. Table V presents the effect of pruning on the total searching time and the total expansions for 200 random on-line queries.

These are the averages taken over 10 different episodes. 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 III can be attributed to the averaging effect of random queries, posing easier exploration problems compared to the selected extreme.

We furthermore observed that geometric cuts perform well in two cases. First, if test data contains many paths to the target, the exploration algorithm is slow, because it does not know which route to take, i.e. there are many possible neighbors that it has to consider. When some of them are excluded, the search space is much smaller. If there is no path at all to the target, bounding boxes can 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 VI 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 VI
EFFECT OF DECOMPRESSION (ON-LINE).

X. GEOMETRIC VISUALIZATION

For visualization of traces and solution paths, we have adapted an $O(n)$ on-line vertex-detection and data reduction algorithms for freehand writings designed for data recording and replay of pen-based inputs. Vertex recognition recognizes changes in the orientation during the execution of a trace. A point is a vertex, if the angle of the curve γ_k at this point is below a certain fixed angle γ and the distance to the two neighboring points. The cosine of the angle $\cos\gamma_k$ is computed in the formula $(p_k - p_{k-1}) \cdot (p_k - p_{k+1})/(|p_k - p_{k-1}| \cdot |p_k - p_{k+1}|)$, thus saving cosine computations when thresholding with $\cos\gamma$.

Another important aspect for the proper representation of lines is spline fitting, so that the points are smoothly connected through polynomials of low degree. To calculate an interpolating spline the `pscurve` command in the LaTeXmacro *PSTricks* [17] is used. The formulae for points $p_1, \ldots, p_n$ require to compute four control points $s_1, \ldots, s_4$ of a cubic spline from p_i to p_{i+1} as follows: $s_0 = p_i$, $s_1 = p_{i-1} + |p_i - p_{i-1}| \cdot d' \cdot m'$, $s_3 = p_{i+1} - |p_i - p_{i+1}| \cdot d \cdot m$, and $s_4 = p_{i+1}$, where d' and m' are the old values of d and m with

$$\begin{aligned} d &= (p_i - p_{i-1})|p_{i+1} - p_i| + (p_{i+1} - p_i)|p_i - p_{i-1}| \\ m &= \frac{\alpha}{2|d|} \left| \cos\left(\frac{a(p_i - p_{i-1}) - a(p_{i+1} - p_i)}{2} \right) \right|^{\beta} \\ a(p) &= \arctan(x/y) \text{ given } p = (x, y). \end{aligned}$$

The parameters are $\alpha = 0.690176$ and $\beta = 0.1$, and at the beginning, $s_1 = p_1$ as well as at the end, $s_2 = p_n$.

Data reduction with splines faces another problem. If we try to calculate the exact distance between a curve with and without the point in question, we have to deal with polynomials of 6th degree, leading to an inefficient algorithm. The following reduction schema approximates the distance to eliminate point p_i as follows: compute the parametric representation of the spline s' through the points p_{i-2}, p_{i-1}, p_{i+1}, and p_{i+1}, compare it to spline s that includes all five points, and omit p_i if the distance s and s' is within a threshold for some certain test set of intermediate points. 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.

Computing the travel graph according to a set of splines can also be an option at all, but calls for refined algorithmic solutions as e.g. addressed in the EXACUS project[11] at MPI, Saarbrücken.

XI. NAVIGATION IN DYNAMIC ENVIRONMENT

Consider the scenario when, while following a shortest path, the user detects that because of a road accident or some other

[11] `www.mpi-sb.mpg.de/projects/EXACUS`

reason, there is a traffic jam. A traffic jam implies an increase in travel time for the affected area. In this case a second shortest path from the current position to the destination is needed that also avoids the affected area.

We analyze two models for introducing dynamics in the system. In the first model, due to a disturbance, we assume that some particular edges are directly affected. We model this situation as an increase in the weights of these edges. Furthermore, we present an alternative approach where a disturbance is represented as a geometrical object affecting the edges covered by that object on the graph. In both of these models we assume that the changes in the graph are temporary in nature, i.e., they disappear after some time.

Increasing the weights of some edges implies the invalidity of some of the bounding boxes, particularly the ones that contain edges with increased edge weights. This restricts the bounding-box searching algorithm to make use of the precomputed information.

A. Dynamic Queries

A dynamic shortest path problem can be defined as a 3-tuple $D = (G_c, C, q)$. It aims at answering the shortest path query q in the graph G_c, in the presence of a set of constraints C. The query q is represented as $q = (s, t, time_s)$ with s as the start node, t as the target node and $time_s$ as the desired starting time when the path is to be traversed.

The set of constraints C defines disturbances that may arise in the road network in the course of time. At the basic level, we define these constraints to be affecting a subset of edges by increasing their weights. Each constraint is also characterized by the time after which the constraint is no longer valid. It corresponds to the situation when the traffic flow has returned to normal after a disturbance.

The increase in weight can be accommodated in the graph G_c by increasing the values $w_c(e)$ of edges e. This restricts us to search the shortest path in the modified graph.

This model can be extended to a more general one that facilitates quickest path searching or a combination of both also. The extension involves utilizing a separate weight function w and a priority parameter τ value to give priority to distance or time. The weight of an edge e is now $w(e) = \tau \times (T_{end}(e) - T_{start}(e)) + (1 - \tau) \times w_c(e)$. Note that this weight function remains constant during the computation.

The constraints can now be defined in terms of an increase in the weight w of an edge. This leads to a re-formalization of our problem as a 5-tuple $D = (G_c, w, \tau, C, q)$.

In the following we elaborate our problem from two different perspectives. In the first one we represent disturbances as affecting the edges directly and increasing their weights. While in the second one we represent a disturbance to be affecting a rectangular area on the graph and in turn affecting all the edges underneath it.

B. Disturbances as Individual Edge Dynamics

In this model, we define C to be a set of constraints where each constraint $c \in C$ maps an edge to a *set* of changes due to disturbances on that edge. Note that we use the term *set* because one disturbance might lead to other disturbances that affect the edge weights differently. As an example, a car accident may lead to a blocked lane and a resulting traffic jam. Removing the traffic jam might not result in the opening of that affected lane.

Formally we define C as a set of constraints $c : E \rightarrow \mathcal{P}\{\Re \times Time\}$, i.e., a constraint is a set of tuples representing the changes in the weight of an edge due to disturbances and the time when the change disappears.

Sometimes it is difficult to pinpoint the exact location of an edge where the disturbance actually is. To accommodate this difficulty we extend our model to allow the definition of constraints on the graph in the form of geometrical objects.

C. Disturbances as Geometrical Objects

For simplicity reasons we assume each geometrical object to be an iso-oriented rectangle. This rectangle is considered to be a separate entity from the graph and is situated in a layer on top of the graph.

All the edges underneath a rectangular object are assumed to be affected by the disturbance. There can actually be three kinds of edges that are affected. One that are completely contained inside the rectangle, one that intersect the boundary of the rectangle at exactly one point, and one that intersect the boundary of the rectangle at two points.

These disturbances can be formalized in the form of constraints $c : \Gamma \rightarrow \Re \times Time$ where Γ is the set of rectangles corresponding to the affected areas. A rectangle $\gamma \in \Gamma$ is mapped to an *increase* in the weight of the edges underneath it and a *Time* value after which that constraint is considered as *void*. Note that Γ can also be viewed as a function that maps a γ to a set $E_\gamma \subseteq E$, i.e. the set of edges affected by γ.

D. Disturbances and Static Information

Due to a change in the weights, the static information kept in the shortest path bounding boxes might become invalid. Some of the bounding boxes may mislead the shortest path algorithm and can result in a path other than the *shortest path*. Re-computing the bounding boxes for every change is too time consuming, since the disturbances are temporary and disappear after a short time. Also, the exploration of a particular query utilizes only a small subset of the problem graph. This gives us the idea to characterize the bounding boxes that may have become invalid and that would be encountered for a particular query, at the start of the exploration. In case those bounding boxes are encountered during exploration, we can simply ignore the pre-computed information and rely on other heuristic methods like A*.

A bounding box can mislead a shortest path algorithm, if it contains at least one edge whose weight has been changed since the bounding boxes were computed. The bounding boxes that contain no affected edges are *valid* and can be used for the pruning of the search space in a shortest path algorithm. We observe that if the bounding boxes of the outgoing edges of the start node S do not contain any of the affected edge i.e., they are valid, the shortest path Π searched in the pruned search space remains to be the shortest path.

E. Searching in Dynamic Environment

We distinguish two approaches for searching for a shortest path in a dynamic environment. The first approach that we called as *Graph-Update* or *Off-line* approach updates the graph with new edge information on the arrival of a disturbance. The search is then performed on the updated graph. The pruning information is used only if the bounding boxes of the outgoing edges of s do not contain any of the affected edge. This test is performed by using a geometric data structure called Segment tree. Segment tree is a data structure for saving parallel line segments and with certain enhancements can be used to answer *window-queries* for arbitrary oriented segments, i.e., giving a rectangular window overlaid on a set of line segments, return the line segments that are intersected by the window. The fully dynamic variant of segment tree support insertion of new segments in $O(\log n)$ time and deletions of segments in $O(\log n \cdot \alpha(i,n))$ time, where $\alpha(i,n)$ is the extremely slowly growing functional inverse of the Ackermann's function. The window-query operation can be carried out in $O(\log^2 n + k)$, where k is the number of reported segments. Segment tree requires an $O(n \log n)$ space [18]. In our case, the query rectanges are the bounding boxes corresponding to the outgoing edges of s and the segment tree is used to maintain the edges whose weight has been changed since the bounding boxes are computed.

For the model when disturbances are represented by rectanges, the initial test can be viewed as red-blue rectangle intersection problem. It deals with computing the intersection points of two different types of rectangles. Using a sweep line algorithm, this problem can be solved in $O(n \log n + k)$ time, where n is the total number of rectangles of both types and k is the number of intersections reported [19].

The second approach that we called as *Exploration-time Checking* or *on-line* approach utilizes the observation that it is possible that some of the constraints have terminated and no longer be there by the time the mobile object will reach the affected area. This observation suggest a search procedure, where we continute exploring using our search algorithm along with the pre-computed information until we reach an affected edge. If we do not encounter any affected edge and reach the target, we declare the encountered path as the shortest path based on the results of Section XI-D. In case we encounter an affected edge, we restart the search procedure without utilizing the pre-computed information.

XII. 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 like [20]. In the context of car navigation, lane-precise maps are inferred. 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. In [21] it is shown how existing electronic maps can be adapted to an electronic base map. For domain-independent trajectory planning however, such an approach is not feasible, because bikers or hikers have much more freedom in choosing their route.

Automated leveling the graph structure has been also addressed by [22] with different optimality preserving algorithms for hierarchical structured graphs. In an application scenario from the field of timetable information in public transport, the work gives a detailed analysis and experimental evaluation of shortest path computations based on multi-level graph decomposition.

An experimental study of the impact of geometric pruning cuts for the setting of train graphs is presented in [23]. In the algorithm portfolio, bounding boxes appear to be superior to annotations of angular sectors. Bounding-box pruning extends early observations of [24], where angular sectors of all shortest paths that pass an considered edge were used.

Similar to pattern databases [25], shortest path bounding boxes are memory intense approximations of shortest path information in the state space graph to be inferred before a query is processed. Pattern databases improve the quality of the heuristic estimate, while bounding boxes improve pruning capability, effectively reducing the branching factor. Moreover, while pattern databases fix the goal state and construct the database with one shortest path backward exploration in abstract space, pruning with bounding boxes stores approximated information to be exploited for different start and goal state queries.

We inserted $O(n \log n)$ as the worst-case run time of Dijkstra's algorithm in the travel graph exploration, by referring to a Fibonacci heap implementation and bounded node degree. As in the current implementation the constructed graph is planar, using the graph separator algorithm of [26] would lead to a theoretically faster algorithm, with linear run time for non-negative edge cost. This reduces the pre-computation time to $O(n^2)$. However, to the best of the authors' knowledge, this algorithm has not been implemented yet. Moreover, the single-shot run time would slow down significantly, if negative edges were allowed. Planarity is affected, if we allow invalid intersection (due to bridges or tunnels). Linear time algorithms for a broader graph classes have mostly be devised for restricted weight functions only [27]. Recent experimental results on shortest path search [28] have also only limited impact on our work, since the authors consider undirected graphs and multiple queries to the same target only.

For the case of search in a dynamic scenario [29] presented an approach where the authors have obtained a speed-up of up to a factor of three in re-computing the bounding boxes in case of a change in the weight of an edge. The main idea behind their approach is to enlarge the bounding boxes of the affected edges. The authors extended the bounding box pruning to bi-directional search algorithms by computing additional bounding boxes for the graph with reversed edges. The approach has shown good results in the case of train graphs. But for large graphs ($n > 100,000$) and where disturbances are frequent, the above mentioned approach is not feasible.

XIII. Conclusion

The area of GPS navigation proves to be a challenge for current planning engines. It subsumes several algorithmic issues from computational geometry in general and AI search in particular, such as autonomous robotics to gather and refine raw data by integrating different input sources e.g. by applying Kalman filtering, vision and compression borrowed from handwriting recognition as well as algorithms to build and query the graph, and known and novel search techniques to speed-up shortest path computations.

We combine GPS data from several sources, as opposed to data obtained from dedicated surveying personnel. Automated processing can be much less expensive. The same is true for the price of GPS systems; within the next few years, most new vehicles will likely have at least one GPS receiver, and wireless technology is rapidly advancing to provide the communication infrastructure. Our route planner 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 experiments highlight the applicability of our approach to cope with growing GPS data sources. We expect that in combination of larger inputs with several millions of raw GPS data points can be dealt with our approach. For even larger sets, statistical clustering and external graph construction algorithms are expected.

Dynamic aspects of GPS route planning are crucial if a server maintains and processes timed geometric information on trace availability, to quickly provide alternative routes to front-end users. Efficient updates to the condensed shortest path information according to environmental change is limited. Static tables and dynamic updates appear to be ambivalent issues. We observe that the increase in cost on one edge due to some disturbance does not change shortest path information of bounding boxes *behind* but *before* the affected area. We presented two models for incorporating dynamics in our system. We observe that employing some efficient computational geometry algorithms, the dynamic aspects of a navigation system can be dealt with easily.

Subsequently, we are very much concerned on improving the scaling behavior and the dynamics of our route planning system. The former issue calls for external shortest path graph search algorithms [30] that exploit the spatial structure to minimize secondary memory accesses [31].

Acknowledgments

Thanks to Lioudmila Belenkaia for links to trace data reduction. The work is supported by the DFG under grants ED 74/2-1 and ED 74/3-1, by the Human Potential Programme of the European Union under contract no. HPRN-CT-1999-00104 (AMORE), and by the DFG under grant WA 654/12-1.

References

[1] A. C. Harvey, *Forecasting, Structural Time Series Models and the Kalman Filter*. Cambridge University Press, 1990.

[2] J. Hershberger and J. Snoeyink, "An $O(n \log n)$ implementation of the Douglas-Peuker algorithm for line simplification," *ACM Computational Geometry*, pp. 383–384, 1994.

[3] D. H. Douglas and T. K. Peuker, "Algorithms for the reduction of the number of points," *The Canadian Cartographer*, vol. 10, pp. 112–122, 1973.

[4] J. L. Bentley and T. A. Ottmann, "Algorithms for reporting and counting geometric intersections," *Transactions on Computing*, vol. 28, pp. 643–647, 1979.

[5] B. Chazelle, "Reporting and counting segment intersections," *Computing System Science*, vol. 32, pp. 200–212, 1986.

[6] B. Chazelle and H. Edelsbrunner, "An optimal algorithm for intersecting lines in the plane," *Journal of the ACM*, vol. 39, pp. 1–54, 1992.

[7] I. J. Balaban, "An optimal algorithm for finding segment intersection," in *ACM Symposium on Computational Geometry*, 1995, pp. 339–364.

[8] G. M. Voronoi, "Novelle application des parametres continus a la theorie des formes quadratiques," *Reine und Angewandte Mathematik*, vol. 133, pp. 97–178, 1907.

[9] S. J. Fortune, "A sweepline algorithm for Voronoi diagrams," *Algorithmica*, pp. 153–174, 1987.

[10] L. J. Guibas and J. Stolfi, "Ruler, compass and computer: the design and analysis of geometric algorithms," *Theoretical Foundations of Computer Graphics*, vol. 40, pp. 111–165, 1988.

[11] D. G. Kirkpatrick, "Optimal search in planar subdivisions," *SIAM Journal of Computing*, vol. 12, no. 1, pp. 28–35, 1983.

[12] L. J. Guibas, D. E. Knuth, and M. Sharir, "Randomized incremental construction of Delaunay and Voronoi diagrams," *Algorithmica*, vol. 7, pp. 381–413, 1992.

[13] J. Pearl, *Heuristics*. Addison-Wesley, 1985.

[14] T. H. Cormen, C. E. Leiserson, and R. L. Rivest, *Introduction to Algorithms*. The MIT Press, 1990.

[15] K. Mehlhorn and S. Näher, *The LEDA Platform of Combinatorial and Geometric Computing*. Cambridge University Press, 1999.

[16] 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)*, 1998, pp. 110–117.

[17] T. van Zandt, "PSTricks: Postscript macros for Generic TeX, Users Guide," 1993.

[18] M. J. van Kreveld and M. H. Overmars, "Union-copy structures and dynamic segment trees," *Journal of the ACM*, vol. 40, no. 3, pp. 635–652, 1997.

[19] L. Arge, O. Procopiuc, S. Ramaswamy, T. Suel, and J. S. Vitter, "Theory and practice of I/O-efficient algorithms for multidimensional batched searching problems (extended abstract)," in *Symposium on Discrete Algorithms*, 1998, pp. 685–694.

[20] S. Edelkamp and S. Schroedl, "Route planning and map inference with global positioning traces," in *Essays dedicated to Thomas Ottmann*, ser. Lecture Notes in Computer Science. Springer, 2003, pp. 128–151.

[21] S. Schroedl, S. Rogers, and C. Wilson, "Map refinement from GPS traces," DaimlerChrysler Research and Technology North America, Palo Alto, CA, Tech. Rep. RTC 6/2000, 2000.

[22] F. Schulz, D. Wagner, and C. Zaroliagis, "Using multi-level graphs for timetable information," in *ALENEX*, 2002, pp. 43–59.

[23] 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)*. Springer, 2003.

[24] F. Schulz, D. Wagner, and K. Weihe, "Dijkstra's algorithm on-line: An empirical case study from public railroad transport," *Journal of Experimental Algorithmics*, vol. 5, no. 12, pp. 110–114, 2000.

[25] J. C. Culberson and J. Schaeffer, "Pattern databases," *Computational Intelligence*, vol. 14, no. 4, pp. 318–334, 1998.

[26] M. R. Henzinger, P. Klein, S. Rao, and S. Subramanian, "Faster shortest-path algorithms for planar graphs," *Journal of Computer and System Sciences*, vol. 55, no. 1, pp. 3–23, 1997.

[27] M. Thorup, "Undirected single-source shortest paths with positive integer weights in linear time," *Journal of the ACM*, vol. 46, pp. 362–394, 1999.

[28] S. Pettie, V. Ramachandran, and S. Sridhar, "Experimental evaluation of a new shortest path algorithm," in *ALENEX*, 2002, pp. 126–142.

[29] D. Wagner, T. Willhalm, and C. Zaroliagis, "Dynamic shortest path containers," in *Proc. Algorithmic Methods and Models for Optimization of Railways 2003*, ser. Electronic Notes in Theoretical Computer Science, A. Marchetti-Spaccamela, Ed., 2003, to appear.

[30] U. Meyer, P. Sanders, and J. Sibeyn, Eds., *Memory Hierarchies*, ser. Lecture Notes in Computer Science. Springer, 2003.

[31] S. Edelkamp and S. Schroedl, "Localizing A*," in *National Conference on Artificial Intelligence (AAAI)*, 2000, pp. 885–890.