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 Freiburg
Institut für Informatik
Georges-Köhler-Allee 51
D-79110 Freiburg
jabbar@informatik.uni-freiburg.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 bound and refined topographic information; for example by the bounding boxes of all shortest paths that start with a given edge.

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.

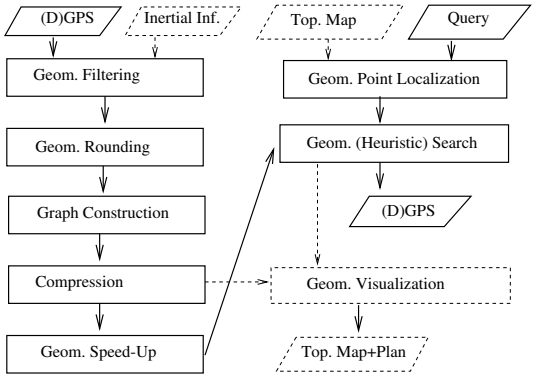

Fig. 1. The architecture of the GPS route-planning system.

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. Nevertheless 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 (possible differential) GPS data is refined by filters that take additional inertial input sources into account.

We refer to the portfolio of algorithms as geometric search, since all computations from trace manipulation to plan visualization exploit the underlying (Euclidean) geometry. For example one geometric speed-up technique we apply, considers shortest path search, where a layout of the trace graph exists that is related to given vertex coordinates. Furthermore, the graph is assumed to be large in the sense that one cannot afford to use more than linear space. In a preprocessing step, for each edge geometric objects 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 localized query to compute a plan, which can be uploaded back to a GPS device for navigation. 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. Related, current and future work is presented in Section VIII followed by the conclusion.

0-7803-8125-4/03/$17.00

II. 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* 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[1] 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 $\log^* n = \min\{k | \underbrace{\log\log\cdots\log}_{k \text{ times}} n = 1\}$.

III. 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.

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

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 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 $l << n$. 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 $G = G_b \cup G_t$, where the nodes V_b in the bottom level b correspond to the original (filtered) GPS data and the nodes in the top level t span the compressed graph G_t.

More precisely, a compression is surjective mapping $\phi : V_b \to V_t$, so that $e_t = (u_t, v_t) \in E_t$ if there exists a path $p = u_b, x_1, \ldots, x_k, v_b$ and $indeg(x_i) = outdeg(x_i) = 1$ with $\phi(u_b) = u_t$ and $\phi(v_b) = v_t$. 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 $\psi : E_t \to E_b$ that when given $e_t = (u, w)$ returns the first hidden edge in the path from u to w.

IV. 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 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.

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 [5].

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.

V. 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 PC or 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 on-line 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 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. 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.

The lower bound that is applied to accelerate route planning is the Euclidean distance $|| \cdot ||_2$, which measures the *flight distance* to the set of goal nodes. The heuristic is consistent by the triangle inequality. A* for consistent estimates with respect to Dijkstra's SSSP 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 *fastest path*, the lower bound has to be modified. This can be done by dividing it by an upper bound for speed.

B. Geometric Speed-Ups

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* 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))
```

In the pseudo-code implementation 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 [6], the space requirements are linear, since no adjacency matrix representation is needed.

Johnson's algorithm [6] shows that pre-computation would also be available in time $O(n^2 \log n)$ even if the edge weights

Fig. 2. Result of a shortest path query for a small bicycle trace.

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

VI. Experiments

We have built our generic system GPS-ROUTE[2] on top of the LEDA algorithm library [7] 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.

Traces (and maps) have to be provided before the route planning algorithm, on which on-line queries are executed. In Figure 2 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.

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.

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

#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. 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.

	#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.

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	#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 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 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).

VII. 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* by [8] 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.

VIII. Related Work

In [9] it is shown how the GPS data can be condensed dramatically by inferring roads and even lanes in the context of car navigation. 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 [10] 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 [11]. In the algorithm portfolio, bounding boxes appear to be superior to annotations of angular sectors. Bounding-box pruning extends early observations of [12], where angular sectors of all shortest paths that pass an considered edge were used.

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 [13] 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 slows 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 [14]. Recent experimental results on shortest path search [15] have also only limited impact on our work, since the authors consider undirected graphs and multiple queries to the same target only.

IX. 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 larger inputs with several millions of raw GPS data points can be dealt with. For even larger sets, statistical clustering and external graph construction algorithms are expected.

We are also concerned on dynamic aspects of GPS route planning with a server maintaining and processing timed geometric information on trace availability, to quickly provide alternative routes to front-end users.

The software 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.

Acknowledgment

Thanks to the DFG for support in the projects Ed 74/3-1 and WA 654/12-1 and to the European Union for support under contract no. HPRN-CT-1999-00104 (AMORE).

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

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

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

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

[9] 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.

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

[11] 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.

[12] 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.

[13] 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.

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

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