Route Planning and Map Inference with Global Positioning Traces

Stefan Edelkamp and Stefan Schrödl

[1] Institut für Informatik
Georges-Köhler-Allee 51
D-79110 Freiburg
edelkamp@informatik.uni-freiburg.de
[2] DaimlerChrysler Research and Technology
1510 Page Mill Road
Palo Alto, CA 94303
schroedl@rtna.daimlerchrysler.com

Abstract. Navigation systems assist almost any kind of motion in the physical world including sailing, flying, hiking, driving and cycling. On the other hand, traces supplied by global positioning systems (GPS) can track actual time and absolute coordinates of the moving objects.
Consequently, this paper addresses efficient algorithms and data structures for the route planning problem based on GPS data; given a set of traces and a current location, infer a short(est) path to the destination.
The algorithm of Bentley and Ottmann is shown to transform geometric GPS information directly into a combinatorial weighted and directed graph structure, which in turn can be queried by applying classical and refined graph traversal algorithms like Dijkstras' single-source shortest path algorithm or A*.
For high-precision map inference especially in car navigation, algorithms for road segmentation, map matching and lane clustering are presented.

1 Introduction

Route planning is one of the most important application areas of computer science in general and graph search in particular. Current technology like hand-held computers, car navigation and GPS positioning systems ask for a suitable combination of mobile computing and course selection for moving objects.

In most cases, a possibly labeled weighted graph representation of all streets and crossings, called the *map*, is explicitly available. This contrasts other exploration problems like puzzle solving, theorem proving, or action planning, where the underlying problem graph is implicitly described by a set of rules.

Applying the standard solution of Dijkstra's algorithm for finding the single-source shortest path (SSSP) in weighted graphs from an initial node to a (set of) goal nodes faces several subtle problems inherent to route planning:

1. Most maps come on external storage devices and are by far larger than main memory capacity. This is especially true for on-board and hand-held computer systems.

2. 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.
3. Maps are likely to be inaccurate and to contain systematic errors in the input sources or inference procedures.
4. It is costly to keep map information up-to-date, since road geometry continuously changes over time.
5. Maps only contain information on road classes and travel distances, which is often not sufficient to infer travel time. In rush hours or on bank holidays, the time needed for driving deviates significantly from the one assuming usual travel speed.
6. In some regions of the world digital maps are not available at all.

The paper is subdivided into two parts. In the first part, it addresses the process of map construction based on recorded data. In Section 2, we introduce some basic definitions. We present the *travel graph inference problem*, which turns out to be a derivate of the output sensitive sweep-line algorithm of Bentley and Ottmann. Subsequently, Section 3 describes an alternative statistical approach. In the second part, we provide solutions to accelerate SSSP computations for time or length optimal route planning in an existing accurate map based on Dijkstra's algorithm, namely A* with the Euclidean distance heuristic and refined implementation issues to deal with the problem of restricted main memory.

2 Travel Graph Construction

Low-end GPS data devices with accuracies of about 2-15 m and mobile data loggers (e.g. in form of palmtop devices) that store raw GPS data entries are nowadays easily accessible and widly distributed. To visualize data in addition to electronic road maps, recent software allows to include and calibrate maps from the Internet or other sources. Moreover, the adaption and visualization of topographical maps is no longer complicated, since high-quality maps and visualization frontends are provided at low price from organizations like *the Surveying Authorities of the States of the Federal Republic of Germany* with the TK50 CD series. Various 2D and 3D user interfaces with on-line and off-line tracking features assist the preparation and the reflection of trips.

In this section we consider the problem of generating a travel graph given a set of traces, that can be queried for shortest paths. For the sake of clarity, we assume that the received GPS data is accurate and that at each inferred crossing of traces, a vehicle can turn into the direction that another vehicle has taken.

With current technology of global positioning systems, the first assumption is almost fulfilled: on the low end, (differential) GPS yields an accuracy in the range of a few meters; high end positioning systems with integrated inertial systems can even achieve an accuracy in the range of centimeters.

The second assumption is at least feasible for hiking and biking in unknown terrain without bridges or tunnels. To avoid these complications especially for car navigation, we might distinguish valid from invalid crossings. Invalid crossing are ones with an intersection angle above a certain threshold and difference in velocity outside a certain interval. Fig. 1 provides a small example of a GPS trace that was collected on a bike on the campus of the computer science department in Freiburg.

```
# latitude, longitude, date (yyyymmdd), time (hhmmss)

        48.0131754,7.8336987,20020906,160241
        48.0131737,7.8336991,20020906,160242
        48.0131720,7.8336986,20020906,160243
        48.0131707,7.8336984,20020906,160244
        48.0131716,7.8336978,20020906,160245
        48.0131713,7.8336975,20020906,160246
```

Figure 1. Small GPS trace.

2.1 Notation

We begin with some formal definitions. *Points* in the plane are elements of $\mathbb{R} \times \mathbb{R}$, and *line segments* are pairs of points. A *timed point* $p = (x, y, t)$ has global coordinates x and y and additional time stamp t, where $t \in \mathbb{R}$ is the absolute time to be decoded in year, month, day, hour, minute, second and fractions of a second. A *timed line segment* is a pair of timed points. A *trace* T is a sequence of timed points $p_1 = (x_1, y_1, t_1), \dots, p_n = (x_n, y_n, t_n)$ such that t_i, $1 \le i \le n$, is increasing. A *timed path* $P = s_1, \dots, s_{n-1}$ is the associated sequence of timed line segments with $s_i = (p_i, p_{i+1})$, $1 \le i < n$. The angle of consecutive line segements on a (timed) path and the velocity on timed line segments are immediate consequences of the above definitions.

The *trace graph* $G_T = (V, E, d, t)$ is a directed graph defined by $v \in V$ if its coordinates (x_v, y_v) are mentioned in T, $e = (u, v) \in E$ if the coordinates of u and v correspond to two successive timed points (x_u, y_u, t_u) and (x_v, y_v, t_v) in T, $d(e) = ||u - v||_2$, and $t(e) = t_v - t_u$, where $||u - v||_2$ denotes the Euclidean distance between (the coordinates of) u and v.

The *travel graph* $G'_T = (V', E', d, t)$ is a slight modification of G_T including its line segment intersections. More formally, let $s_i \cap s_j = r$ denote that s_i and s_j intersect in point p, and let $I = \{(r, i, j) \mid s_i \cap s_j = r\}$ be the set of all intersections, then $V' = V \cup \{r \mid (r, i, j) \in I\}$ and $E' = E \cup E_a \setminus E_d$, where $E_d = \{(s_i, s_j) \mid \exists r : (r, i, j) \in I\}$, and $E_a = \{(p, r), (r, q), (p', r), (r, q') \in V' \times V' \mid (r, i, j) \in I \text{ and } r = (s_i = (p, q) \cap s_j = (p', q'))\}$. Note that intersection points r have no time stamp. Once more, the new cost values for $e = (u, v) \in E' \setminus E$ are determined by $d(e) = ||u - v||_2$, and by $t(e) = t(e')d(e)/d(e')$ with respect to the original edge $e' \in E_d$. The latter definition of time assumes a uniform speed on every line segment, which is plausible on sufficiently small line segments.

The *travel graph* G'_D of a set D of traces $T_1, \dots T_l$ is the travel graph of the union graph of the respective trace graphs $G_{T_1}, \dots, G_{T_k}$, Where the union graph $G = (V, E)$ of two graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ is defined as $V = V_1 \cup V_2$ and $E = E_1 \cup E_2$.

For the sake of simplicity, we assume that all crossings are in *general position*, so that not more than two line segments intersect in one point. This assumption is not a severe restriction, since all algorithms can be adapted to the more general case. We also might exclude matching endpoints from the computation, since we already know that two consecutive line segments intersect at the recorded data point. If a vehicle stops while the GPS recorder is running, zero-length sequences with strictly positive time delays are generated. Since zero-length segments cannot yield crossings, the problem of self loops might be dealt with ignoring these segments for travel graph generation and a re-inserting them afterwards to allow timed shortest path queries.

2.2 Algorithm of Bentley and Ottmann

The plane-sweep algorithm of Bentley and Ottmann [3] infers an undirected planar graph representation (the *arrangement*) of a set of segments in the plane and their intersections. The algorithm is one of the most innovative schemes both from a conceptual and from a algorithmical point of view.

From a conceptional point of view it combines the two research areas of *computational complexity* and *graph algorithms*. The basic principle of an imaginary *sweep-line* that stops on any interesting event is one of the most powerful technique in geometry e.g. to directly compute the Voronoi diagram on a set of n points in optimal time $O(n \log n)$, and is a design paradigm for solving many combinatorial problems like the minimum and maximum in a set of values in the optimal number of comparisons, or the maximum sub-array sum in linear time with respect to the number of elements.

From an algorithmical point of view the algorithm is a perfect example of the application of balanced trees to reduce the complexity of an algorithm. It is also the first *output-sensitive* algorithm, since its time complexity $O((n+k)\log n)$ is measured in both the input and the output length, due to the fact that n input segments may give rise to $k = O(n^2)$ intersections.

The core observation for route planning is that, given a set of traces D in form of a sequence of segments, the algorithm can easily be adapted to compute the corresponding travel graph G'_D. In difference to the original algorithm devised for computational geometry problems, the generated graph structure has to be directed. The direction of each edge e as well as its distance $d(e)$ and travel time $t(e)$ is determined by the two end nodes of the segment. This includes intersections: the newly generated edges inherit direction, distance and time from the original end points.

In Fig. 2 we depicted a snapshot of the animated execution of the algorithm in the client-server visualization Java frontend VEGA [15] *i*) on a line segment sample set and *ii*) on an extended trail according to Fig. 1. The sweep-line proceeds from left to right, with the completed graph to its left.

The algorithm utilizes two data structures: the *event queue* and the *status structure*. In the event queue the active points are maintained, ordered with respect to their x-coordinate. In the status structure the active set of segments with respect to the sweep line is stored in y-ordering. At each intersection the ordering of segments in the status structure may change. Fortunately, the ordering of segments that participate in the intersections simply reverses, allowing fast updates in the data structure. After new neighboring segments are found, their intersections are computed and inserted into the event queue. The abstract data structure needed for implementation are a priority queue for the event queue and a search tree with neighboring information for the status data structure. Using a standard heap for the former and a balance tree for the latter implementation yields an $O((n+k)\log n)$ time algorithm.

The lower bound of the problem's complexity is $\Omega(n \log n + k)$ and the first step to improve time performance was $O(n \log^2 n / \log\log n + k)$ [5]. The first $O(n \log n + k)$

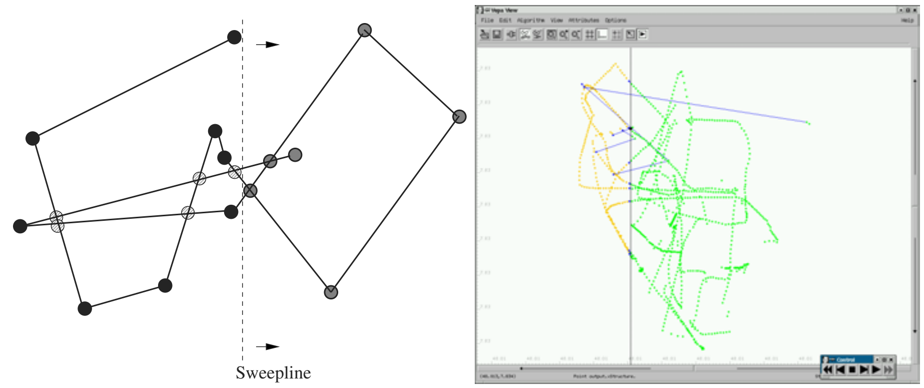

Figure 2. Visualization of the sweep-line algorithm of Bentley and Ottmann on a *i*) tiny and *ii*) small data set in the client-server visualization Java frontend VEGA.

algorithm [6] used $O(n+k)$ storage. The $O(n\log n+k)$ algorithm with $O(n)$ space is due to Balaban [2].

For trace graphs the $O((n+k)\log n)$ implementation is sufficiently fast in practice. As a small example trace file consider $n = 2^{16} = 65,536$ segment end points with $k = 2^8 = 256$ intersections. Then $n^2 = 2^{32} = 4,294,967,296$, while $(n+k)\log n = (2^{16}+2^8)\cdot 16 = 1,052,672$ and $n\log n + k = 1,048,832$.

3 Statistical Map Inference

Even without map information, on-line routing information is still available, e.g. a driving assistance system could suggest *you are 2 meters off to the right of the best route*, or *you have not followed the suggested route, I will recompute the shortest path from the new position*, or *turn left in about 100 meter in a resulting angle of about 60 degrees*.

Nevertheless, route planning in trace graphs may have some limitations in the presentation of the inferred route to a human, since abstract maps compact information and help to adapt positioning data to the real-world.

In this section, an alternative approach to travel graphs is presented. It concentrates on the map inference and adaptation problem for car navigation only, probably the most important application area for GPS routing. One rationale in this domain is the following. Even in one lane, we might have several traces that overlap, so that the number of line segment intersections k can increase considerably. Take $m/2$ parallel traces on this lane that intersect another parallel $m/2$ traces on the lane a single lane in a small angle, then we expect up to $\Theta(m^2)$ intersections in the worst case.

We give an overview of a 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. Our approach is a statistical one: we combine 'lots of bad' GPS data from a fleet of vehicles, as opposed to 'few but highly accurate' data obtained from dedicated surveying vehicles operated by specially trained personnel. Automated processing can be much less expensive. The same is true for the price of DGPS systems; within the next few years, most new vehicles will likely have at least one DGPS receiver, and wireless technology is rapidly advancing to provide the communication infrastructure. The result will be more accurate, cheaper, up-to-date maps.

3.1 Steps in the Map Refinement Process

Currently commercially available digital maps are usually represented as graphs, where the nodes represent intersections and the edges are unbroken road *segments* that connect the intersections. Each segment has a unique identifier and additional associated attributes, such as a number of *shape points* roughly approximating its geometry, the road type (e.g., highway, on/off-ramp, city street, etc), speed information, etc. Generally, no information about the number of lanes is provided. The usual representation for a a two-way road is by means of a single segment. In the following, however, we depart from this convention and view segments as unidirectional links, essentially splitting those roads in two segments of opposite direction. This will facilitate the generation of the precise geometry.

The task of map refinement is simplified by decomposing it into a number of successive, dependent processing steps. Traces are divided into subsections that correspond to the road segments as described above, and the geometry of each individual segment is inferred separately. Each segment, in turn, comprises a subgraph structure capturing its lanes, which might include splits and merges. We assume that the lanes of a segment are mostly parallel. In contrast to commercial maps, we view an intersection as a structured region, rather than a point. These regions limit the segments at points where the traces diverge and consist of unconstrained trajectories connecting individual lanes in adjacent segments.

The overall map refinement approach can be outlined as follows.

1. Collect raw DGPS data (traces) from vehicles as they drive along the roads. Currently, commercially available DGPS receivers output positions (given as longitude/latitude/altitude coordinates with respect to a reference ellipsoid) at a regular frequency between 0.1 and 1 Hz.
   Optionally, if available, measurements gathered for the purpose of electronic safety systems (anti-lock brakes or electronic stability program), such as wheel speeds and accelerometers, can be integrated into the positioning system through a *Kalman filter* [14]. In this case, the step 2 (filtering or smoothing) can be accomplished in the same procedure.
2. Filter and resample the traces to reduce the impact of DGPS noise and outliers. If, unlike in the case of the Kalman filter, no error estimates are available, some of the errors can be detected by additional indicators provided by the receiver, relating to satellite geometry and availability of the differential signal; others (e.g., so-called *multipath* errors) only from additional plausibility tests, e.g., maximum acceleration according to a vehicle model. Resampling is used to balance out the bias of traces recorded at high sampling rates or at low speed. Details of the preprocessing are beyond the scope of the current paper and can be found in a textbook such as [23].

3. Partition the raw traces into sequences of segments by *matching* them to an initial base map. This might be a commercial digital map, such as that of Navigation Technologies, Inc. [22]. Section 3.2 presents an alternative algorithm for inferring the network structure from scratch, from a set of traces alone.

   Since in our case we are not constrained to a real-time scenario, it is useful to consider the context of sample points when matching them to the base map, rather than one point at a time. We implemented a map matching module that is based on a modified best-first path search algorithm based on the Dijkstra-scheme [9], where the matching process compares the DGPS points to the map shape points and generates a cost that is a function of their positional distance and difference in heading. The output is a file which lists, for each trace, the traveled segment IDs, along with the starting time and duration on the segment, for the sequence with minimum total cost (a detailed description of map matching is beyond the scope of this paper).
4. For each segment, generate a *road centerline* capturing the accurate geometry that will serve as a reference line for the lanes, once they are found. Our spline fitting technique will be described in Section 3.3.
5. Within each segment, cluster the perpendicular offsets of sample points from the road centerline to identify *lane* number and locations (cf. Section 3.4).

3.2 Map Segmentation

In the first step of the map refinement process, traces are decomposed into a sequence of sections corresponding to road segments. To this end, an initial base map is needed for map matching. This can either be a commercially available map, such as that of Navigation Technologies, Inc. [22]; or, we can infer the connectivity through a spatial clustering algorithm, as will be described shortly.

These two approaches both have their respective advantages and disadvantages. The dependence on a commercial input map has the drawback that, due to its inaccuracies (Navigation Technologies advertises an accuracy of 15 meters), traces sometimes are incorrectly assigned to a nearby segment. In fact, we experienced this problem especially in the case of highway on-ramps, which can be close to the main lanes and have similar direction.

A further disadvantage is that roads missing in the input map cannot be learned at all. It is impossible to process regions if no previous map exists or the map is too coarse, thus omitting some roads.

On the other hand, using a commercial map as the initial baseline associates additional attributes with the segments, such as road classes, street names, posted speeds, house numbers, etc. Some of these could be inferred from traces by related algorithms on the basis of average speeds, lane numbers, etc. Pribe and Rogers [25] describe an approach to learning traffic controls from GPS traces. An approach to travel time prediction is presented in [13]. However, obviously not all of this information can be independently recovered. Moreover, with the commercial map used for segmentation, the refined map will be more compatible and comparable with applications based on existing databases.

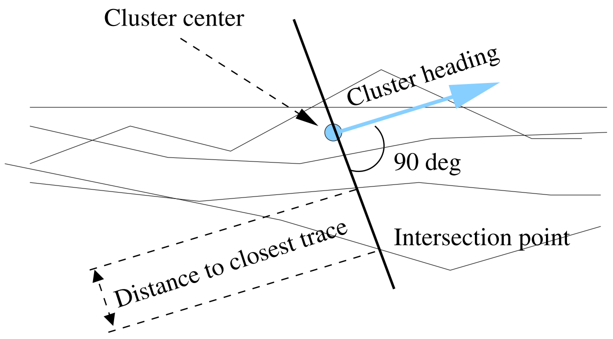

Figure 3. Distance between candidate trace and cluster seed

Road Segment Clustering In this section, we outline an approach to *inferring* road segments from a set of traces simultaneously. Our algorithm can be regarded as a *spatial clustering* procedure. This class of algorithms is often applied to recognition problems in image processing (See e.g. [10] for an example of road finding in aerial images). In our case, the main questions to answer are to identify common segments used by several traces, and to locate the branching points (intersections). A procedure should be used that exploits the contiguity information and temporal order of the trace points in order to determine the connectivity graph. We divide it into three stages: cluster seed location, seed aggregation into segments, and segment intersection identification.

Cluster Seed Location *Cluster seed location* means finding a number of sample points on different traces belonging to the same road. Assume we have already identified a number of trace points belonging to the same cluster; from these, a mean values for position and heading is derived. In view of the later refinement step described in Section 3.3, we can view such a cluster center as one point of the *road centerline*.

Based on the assumption of lane parallelism, we measure the distance between traces by computing their intersection point with a line through the cluster center that runs perpendicular to the cluster heading; this is equivalent to finding those points on the traces whose projection onto the tangent of the cluster coincides with the cluster location, see Fig. 3.

Our similarity measure between a new candidate trace and an existing cluster is based both on its minimum distance to other member traces belonging to the cluster, computed as described above; and on the difference in heading. If both of these indicators are below suitable thresholds (call them θ and δ, respectively) for two sample points, they are deemed to belong to the same road segment.

The maximum heading difference δ should be chosen to account for the accuracy of the data, such as to exclude sample points significantly above a high quantile of the error distribution. If such an estimate is unavailable, but a number of traces have already been found to belong to the cluster, the standard deviation of these members can give a clue. In general we found that the algorithm is not very sensitive to varation of δ.

The choice of θ introduces a trade-off between two kinds of segmentation errors: if it is too small, wide lanes will be regarded as different roads; in the opposite case,

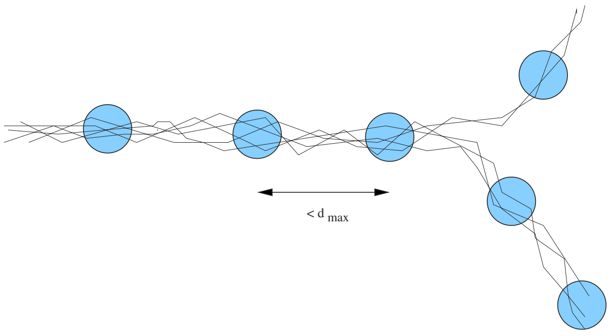

Figure 4. Example of traces with cluster seeds

nearby roads would be identified. In any case, the probablility that the GPS error exceeds the difference between the distance to the nearest road and the lane width is a lower bound for the segmentation error. A suitable value depends on the expected GPS errors, characteristics of the map (e.g., the relative frequencies of four-way intersections vs. freeway ramps), and also on the relative risks associated with both tyes of errors which are ultimately determined by the final application. As a conservative lower bound, θ should be at least larger than the maximum lane width, plus a tolerance (estimated standard deviation) for driving off the center of the lane, plus a considerable fraction of an estimated standard deviation of the GPS error. Empirically, we found the results with values in the range of 10–20 meters to be satisfying and sufficiently stable.

Using this similarity measure, the algorithm now proceeds in a fashion similar to the k-means algorithm [19]. First, we initialize the cluster with some random trace point. At each step, we add the closest point on any of the traces not already in the cluster, unless θ or δ is exceeded. Then, we recompute the average position and heading of the cluster center. Due to these changes, it can sometimes occur that trace points previously contained in the cluster do no longer satisfy the conditions for being on the same road; in this case they are removed. This process is repeated, until no more points can be added.

In this manner, we repeatedly generate cluster seeds at different locations, until each trace point has at least one of them within reach of a maximum distance threshold d_{max}. This threshold should be in an order of magnitude such that we ensure not to miss any intersection (say, e.g., 50 meters). A simple greedy strategy would follow each trace and add a new cluster seed at regular intervals of length d_{max} when needed. An example section of traces, together with the generated cluster centers, are shown in Fig. 4.

Segment Merging The next step is to *merge* those ones of the previously obtained cluster centers that belong to the same road. Based on the connectivity of the traces, two such clusters C_1 and C_2 can be characterized in that (1) w.l.o.g. C_1 precedes C_2, i.e., all the traces belonging to C_1 subsequently pass through C_2, and (2) all the the traces belonging to C_2 originate from C_1. All possible adjacent clusters satisfying this condition are merged. A resulting maximum chain of clusters $C_1, C_2, \dots, C_n$ is called a *segment*, and C_1 and C_n are called the *boundary clusters* of the segment.

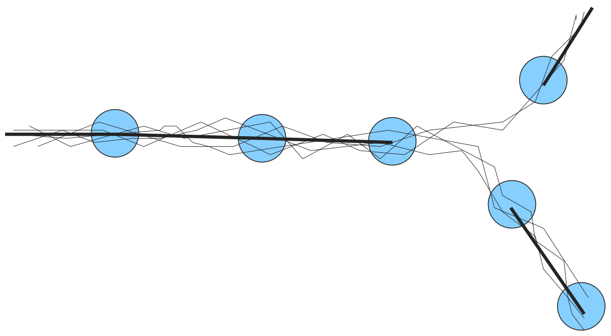

Figure 5. Merged cluster seeds result in segments

At the current segmentation stage of the map refinement process, only a crude geometric representation of the segment is sufficient; its precise shape will be derived later in the road centerline generation step (Section 3.3). Hence, as an approximation, adjacent cluster centers can either be joined by straight lines, polynomials, or one representative trace part (in our algorithms, we chose the latter possibility). In Fig. 5, the merged segments are connected with lines.

Intersections The only remaining problem is now to represent intersections. To capture the extent of an intersection more precisely, we first try to advance the boundary clusters in the direction of the split- or merge zone. This can be done by selecting a point from each member trace at the same (short) travel distance away from the respective sample point belonging to the cluster, and then again testing for contiguity as described above. We extend the segment iteratively in small increments, until the test fails.

The set of adjacent segments of an intersection is determined by (1) selecting all outgoing segments of one of the member boundary clusters; (2) collecting all incoming segments of the segments found in (1); and iterating these steps until completion. Each adjacent segment should be joined to each other segment for which connecting traces exist.

We utilize the concept of a *snake* borrowed from the domain of image processing, i.e., a contour model that is fit to (noisy) sample points. In our case, a simple star-shaped contour suffices, with the end points held fixed at the boundary cluster centers of the adjacent segments. Conceptually, each sample points exert an attracting force on it closest edge. Without any prior information on the shape of the intersection, we can define the ’energy’ to be the sum of the squared distances between each sample point and the closest point on any of the edges, and then iteratively move the center point in an EM-style fashion in order to minimize this measure. The dotted lines in Fig. 6 correspond to the resulting snake for our example.

Dealing with Noisy Data Gaps in the GPS receiver signal can be an error source for the road clustering algorithm. Due to obstructions, it is not unusual to find gaps in the data that span a minute. As a result, interpolation between distant points is not reliable.

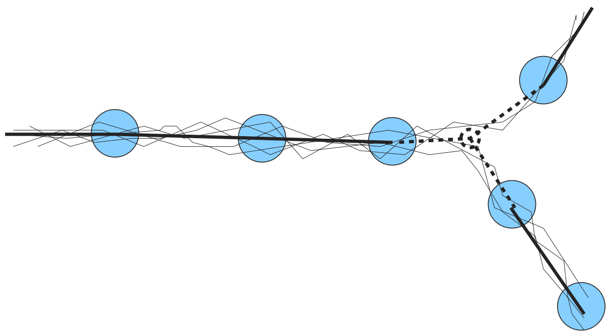

Figure 6. Traces, segments, and intersection contour model (dotted)

As mentioned above, checking for parallel traces crucially depends on *heading* information. For certain types of positioning systems used to collect the data, the heading might have been determined from the direction of differences between successive sample points. In this case, individual outliers, and also lower speeds, can lead to even larger errors in direction.

Therefore, in the first stage of our segmentation inference algorithm, filtering is performed by disregarding trace segments in cluster seeds that have a gap within a distance of d_{max}, or fall outside a 95 percent interval in the heading or lateral offset from the cluster center. Cluster centers are recomputed only from the remaining traces, and only they contribute to the subsequent steps of merging and intersection location with adjacent segments.

Another issue concerns the start and end parts of traces. Considering them in the map segmentation could introduce segment boundaries at each parking lot entrance. To avoid a too detailed breakup, we have to disregard initial and final trace sections. Different heuristics can be used; currently we apply a combined minimum trip length/speed threshold.

3.3 Road Centerline Generation

We now turn our attention to the refinement of individual segments. The *road centerline* is a geometric construct whose purpose is to capture the road geometry. The road centerline can be thought of as a weighted average trace, hence subject to the relative lane occupancies, and not necessarily a trajectory any single vehicle would ever follow. We assume, however, the lanes to be parallel to the road centerline, but at a (constant) perpendicular offset. For the subsequent lane clustering, the road centerline helps to cancel out the effects of curved roads.

For illustration, Fig. 7 shows a section of a segment in our test area. The indicated sample points stem from different traces. Clearly, by comparison, the shape points of the respective NavTech segment exhibit a systematic error. The centerline derived from the sample points is also shown.

It is useful to represent our curves in *parametric form*, i.e., as a vector of coordinate variables $C(u) = (x,y,z)(u)$ which is a function of an independent parameter u, for

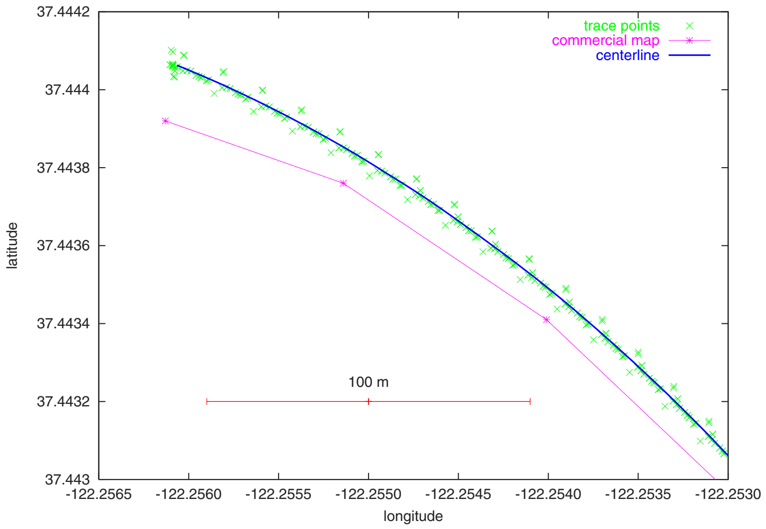

Figure 7. Segment part: NavTech map (bottom), trace points (crosses), and computed centerline

$0 \leq u \leq 1$. The centerline is generated from a set of sample points using a *weighted least squares fit*. More precisely, assume that $Q_0, \ldots, Q_{m-1}$ are the m data points given, $w_0, \ldots, w_{m-1}$, are associated weights (dependent on an error estimate), and $\bar{u}_0, \ldots, \bar{u}_{m-1'}$ their respective parameter values. The task can be formulated as finding a parametric curve $C(u)$ from a class of functions $\mathcal{S}$ such that the Q_k are approximated in the weighted least square sense, i.e.

$$s := \sum_{k=0}^{m-1} w_k \cdot \|Q_k - C(\bar{u}_k)\|^2$$

in a minimum with respect to $\mathcal{S}$, where $\|\cdot\|$ denotes the usual Euclidean distance (2-norm). Optionally, in order to guarantee continuity across segments, the algorithm can easily be generalized to take into account derivatives; if heading information is available, we can use coordinate transformation to arrive at the desired derivative vectors.

The class $\mathcal{S}$ of approximating functions is composed of rational *B-Splines*, i.e., piecewise defined polynomials with continuity conditions at the joining knots (for details, see [24; 27]). For the requirement of continuous curvature, the degree of the polynomial has to be at least three.

If each sample point is marked with an estimate of the measurement error (standard deviation), which is usually available from the receiver or a Kalman filter, then we can use its inverse to weight the point, since we want more accurate points to contribute more to the overall shape.

The least squares procedure [24] expects the *number of control points n* as input, the choice of which turns out to be critical. The control points define the shape of the spline, while not necessarily lying on the spline themselves. We will return to the issue of selecting an adequate number of control points in Section 3.3.

Choice of Parameter Values for Trace Points For each sample point Q_k, a parameter value $\bar{u}_k$ has to be chosen. This parameter vector affects the shape and parameterization of the spline. If we were given a single trace as input, we could apply the widely used *chord length* method as follows. Let d be the total chord length $d = \sum_{k=1}^{m-1} |Q_k - Q_{k-1}|$. Then set $\bar{u}_0 = 0$, $\bar{u}_{m-1} = 1$, and $\bar{u}_k = \bar{u}_{k-1} + \frac{|Q_k - Q_{k-1}|}{d}$ for $k = 1, \ldots, m-2$. This gives a good parameterization, in the sense that it approximates a *uniform* parameterization proportional to the arc length.

For a set of k distinct traces, we have to impose a common ordering on the combined set of points. To this end, we utilize an initial rough approximation, e.g., the polyline of shape points from the original NavTech map segment s; if no such map segment is available, one of the traces can serve as a rough baseline for projection. Each sample point Q_k is *projected* onto s, by finding the closest interpolated point on s and choosing $\bar{u}_k$ to be the chord length (cumulative length along this segment) up to the projected point, divided by the overall length of s. It is easy to see that for the special case of a single trace identical to s, this procedure coincides with the chord length method.

Choice of the Number of Control Points The number of control points n is crucial in the calculation of the centerline; for a cubic spline, it can be chosen freely in the valid range $[4, m-1]$. Fig. 8 shows the centerline for one segment, computed with three different parameters n.

Note that a low number of control points may not capture the shape of the centerline sufficiently well ($n = 4$); on the other hand, too many degrees of freedom causes the result to “fit the error”. Observe how the spacing of sample points influences the spline for the case $n = 20$.

From the latter observation, we can derive an upper bound on the number of control points: it should not exceed the average number of sample points per trace, multiplied by a small factor, e.g., $2 * m/k$.

While the appropriate number of control points can be easily estimated by human inspection, its formalization is not trivial. We empirically found that two measures are useful in the evaluation.

The first one is related to the *goodness of fit*. Averaging the absolute offsets of the sample points from the spline is a feasible approach for single-lane roads, but otherwise depends on the number and relative occupancies of lanes, and we do not expect this offset to be zero even in the ideal case. Intuitively, the centerline is supposed to stay roughly in the middle between all traces; i.e., if we project all sample points on the centerline, and imagine a fixed-length window moving along the centerline, then the average offset of all sample points whose projections fall into this window should be near to zero. Thus, we define the *approximation error* ε_{fit} as the average of these offsets over all windows.

The second measure checks for overfitting. As illustrated in Fig. 8, using a large number of control points renders the centerline “wiggly”, i.e., tends to increase the

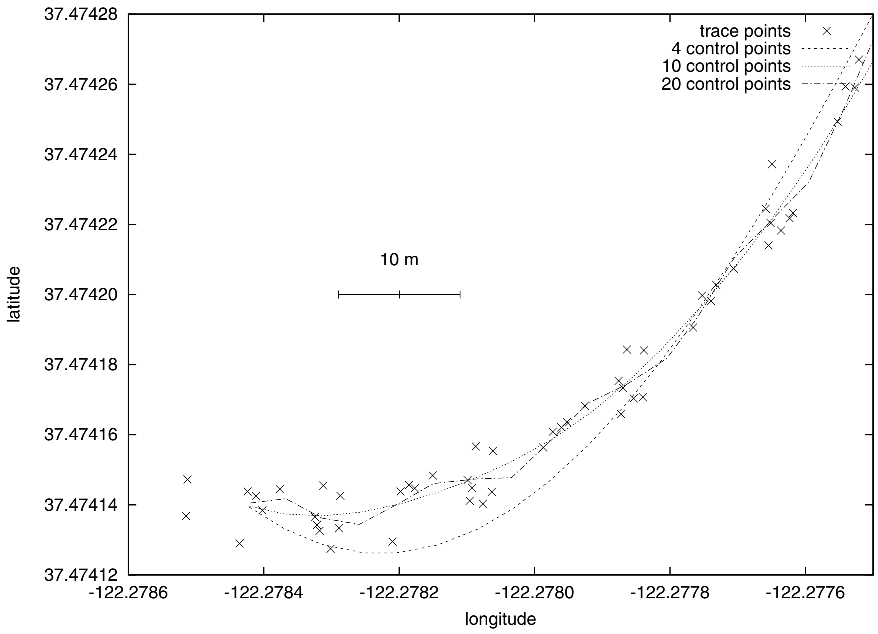

Figure 8. Trace points and centerlines computed with varying number of control points

curvature and makes it change direction frequently. However, according to construction guidelines, roads are designed to be piecewise segments of either straight lines or circles, with clothoids between as transitions. These geometric concepts constrain the curvature to be *piecewise linear*. As a consequence, the second derivative of the curvature is supposed to be zero nearly everywhere, with the exception of the segment boundaries where it might be singular. Thus, we evaluate the curvature of the spline at constant intervals and numerically calculate the second derivative. The average of these values is the *curvature error* $\varepsilon_{\text{curv}}$.

Fig. 9 plots the respective values of ε_{fit} and $\varepsilon_{\text{curv}}$ for the case of Fig. 8 as a function of the number of control points. There is a tradeoff between ε_{fit} and $\varepsilon_{\text{curv}}$; while the former tends to decrease rapidly, the latter increases. However, both values are not completely monotonic.

Searching the space of possible solutions exhaustively can be expensive, since a complete spline fit has to be calculated in each step. To save computation time, the current approach heuristically picks the largest valid number of control points for which $\varepsilon_{\text{curv}}$ lies below an acceptable threshold.

3.4 Lane Finding

After computing the approximate geometric shape of a road in the form of the road centerline, the aim of the next processing step is to infer the number and positions

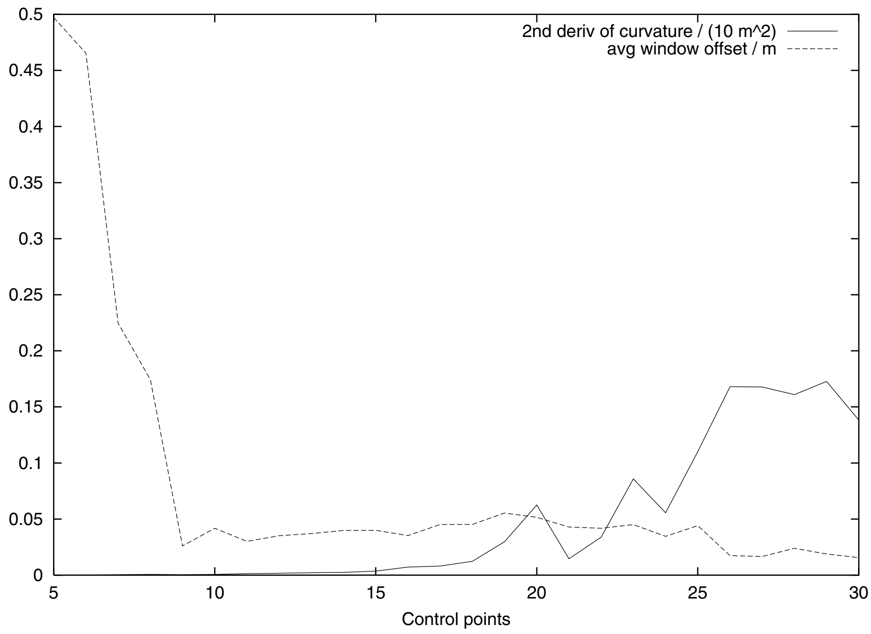

Figure 9. Error Measures ε_{fit} and $\varepsilon_{\text{curv}}$ vs Number of Control Points

of its *lanes*. The task is simplified by canceling out road curvature by the following transformation. Each trace point P is *projected* onto the centerline for the segment, i.e., its nearest interpolated point P' on the map is determined. Again, the arc length from the first centerline point up to P' is the *distance along the segment*; the distance between P and P' is referred to as its *offset*. An example of the transformed data is shown in Fig. 10.

Intuitively, clustering means assigning n data points in a d-dimensional space to k clusters such that some distance measure within a cluster (i.e., either between pairs of data belonging to the same cluster, or to a cluster center) is minimized (and is maximized between different clusters). For the problem of lane finding, we are considering points in a plane representing the flattened face of the earth, so the Euclidean distance measure is appropriate.

Since clustering in high-dimensional spaces is computationally expensive, methods like the *k-means* algorithm use a hill-climbing approach to find a (local) minimum solution. Initially, k cluster centers are selected, and two phases are iteratively carried out until cluster assignment converges. The first phase assigns all points to their nearest cluster center. The second phase recomputes the cluster center based on the respective constituent points (e.g., by averaging) [19].

Segments with Parallel Lanes If we make the assumption that lanes are parallel over the entire segment, the clustering is essentially one-dimensional, taking only into account the offset from the road centerline. In our previous approach [26], a hierarchical

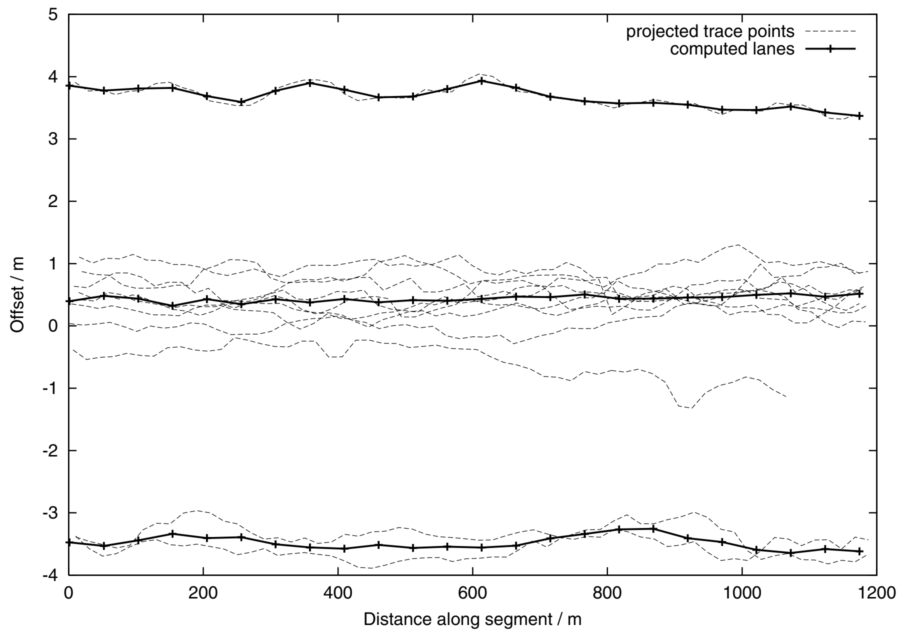

Figure 10. Traces projected on centerline (dashed), computed lane centers (solid)

agglomerative clustering algorithm (*agglom*) was used that terminated when the two closest clusters were more than a given distance apart (which represented the maximum width of a lane). However, this algorithm requires $O(n^3)$ computation time. More recently, we have found that it is possible to explicitly compute the *optimal* solution in $O(k \cdot n^2)$ time and $O(n)$ space using a dynamic programming approach, where n denotes the number of sample points, and k denotes the maximum number of clusters [27].

Segments with Lane Splits and Merges We have previously assumed the ideal case of a constant number of lanes at constant offsets from the road centerline along the whole segment. However, lane widths may gradually change; in fact, they usually get wider near an intersection. Moreover, new lanes can start at any point within a segment (e.g., turn lanes), and lanes can merge (e.g., on-ramps). Subsequently, we present two algorithms that can accommodate the additional complexity introduced by lane merges and splits.

One-dimensional Windowing with Matching One solution to this problem is to augment the one-dimensional algorithm with a windowing approach. We divide the segment into successive windows with centers at constant intervals along the segment. To minimize discontinuities, we use a Gaussian convolution to generate the windows. Each window is clustered separately as described above, i.e., the clustering essentially remains one-dimensional. If the number of lanes in two adjacent windows remains the same, we can associate them in the order given. For lane splits and merges, however, lanes in adjacent windows need to be *matched*.

To match lanes across window boundaries, we consider each trace individually (the information as to which trace each data point belongs to has not been used in the centerline generation, nor the lane clustering). Following the trajectory of a given trace through successive windows, we can classify its points according to the computed lanes. Accumulating these counts over all traces yields a matrix of *transition frequencies* for any pair of lanes from two successive windows. Each lane in a window is matched to that lane in the next window with maximum transition frequency.

3.5 Experimental Results

We have now completed our account of the individual steps in the map refinement process. In this section, we report on experiments we ran in order to evaluate the learning rate of our map refinement process. Our test area in Palo Alto, CA, covered 66 segments with a combined length of approximately 20 km of urban and freeway roads of up to four lanes, with an average of 2.44 lanes.

One principal problem we faced was the availability of a ground truth map with lane-level accuracy for comparison. Development of algorithms to integrate vision-based lane tracker information is currently under way. Unfortunatly, however, these systems have errors on their own and therefore cannot be used to gauge the accuracy of the pure position-based approach described in this article. Therefore, we reverted to the following procedure. We used a high-end real-time kinematic carrier phase DGPS system to generate a base map with few traces [32]. According to the announced accuracy of the system of about 5 cm, and after visual inspection of the map, we decided to define the obtained map as our baseline. Specifically, the input consisted of 23 traces at different sampling rates between 0.25 and 1Hz.

Subsequently, we artificially created more traces of lower quality by adding varying amounts of gaussian noise to each individual sample position ($\sigma = 0.5 \ldots 2$ m) of copies of the original traces. For each combination of error level and training size, we generated a map and evaluated its accuracy.

Fig. 11 shows the resulting error in the *number of lanes*, i.e., the proportion of instances where the number of lanes in the learned map differs from the number of lanes in the base line map at the same position. Obviously, for a noise level in the range of more than half a lane width, it becomes increasingly difficult to distinguish different clusters of road centerline offsets due to overlap. Therefore, the accuracy of the map for the input noise level of $\sigma = 2$ m is significantly higher than that of the lower ones. However, based on the total spread of the traces, the number of lanes can still be estimated. For $n = 253$ traces, their error is below 10 percent for all of them. These remaining differences arise mainly from splits and merges, where in absence of the knowledge of lane markings it is hard to determine the exact branch points, whose position can heavily depend on single lane changes in traces.

Fig. 12 plots the mean absolute difference of the offsets of corresponding lanes between the learned map and the base map, as a function of the training size (number of traces). Again, the case $\sigma = 2$ m needs significantly more training data to converge. For

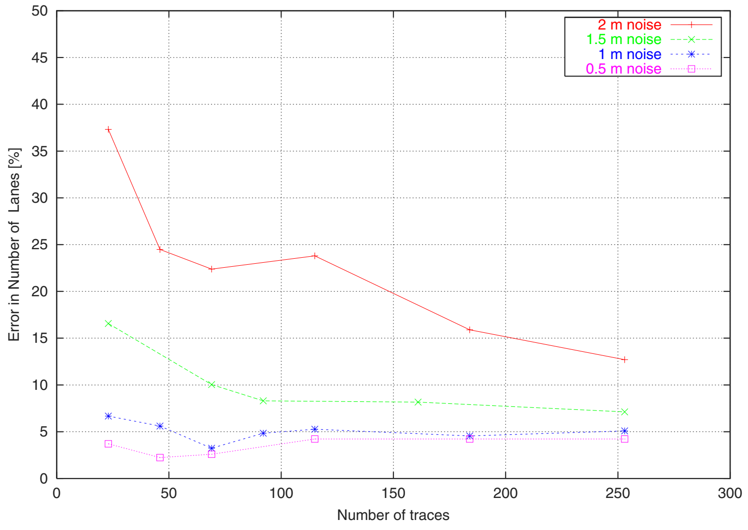

Figure 11. Error in determining the number of lane clusters

$\sigma <= 1.5$ m, the lane offset error decreases rapidly; it is smaller than 15 centimeters after $n = 92$ traces, and thus in the range of the base map accuracy.

4 Searching the Map Graph

Let us now turn our attention to the map usage in on-line routing applications. In some sense, both maps and travel graphs can be viewed as embeddings of weighted general graphs. Optimal paths can be searched with respect to accumulated shortest time t or distance d or any combination of them. We might assume a linear combination for a total weight function $w(u,v) = \lambda \cdot t(u,v) + (1-\lambda) \cdot d(u,v)$ with parameter $\lambda \in \mathbb{R}$ and $0 \leq \lambda \leq 1$.

4.1 Algorithm of Dijkstra

Given a weighted graph $G = (V,E,w)$, $|V| = n$, $|E| = e$, the shortest path between two nodes can be efficiently computed by Dijkstra's single source shortest path (SSSP) algorithm [9].

Table 1 shows a implementation of Dijkstra's algorithm for implicitly given graphs that maintains a visited list *Closed* in form of a hash table and a priority queue of the nodes to be expanded, ordered with respect to increasing merits f.

The run time of Dijkstra's algorithm depends on the priority queue data structure. The original implementation of Dijkstra runs in $O(n^2)$, the standard textbook algorithm in $O((n+e)\log n)$ [7], and utilizing Fibonacci-heaps we get $O(e+n\log n)$ [12]. If the weights are small, buckets are preferable. In a Dial the i-th bucket contains all elements with a f-value equal to i [8]. Dials yields $O(e+n \cdot C)$ time for SSSP, with

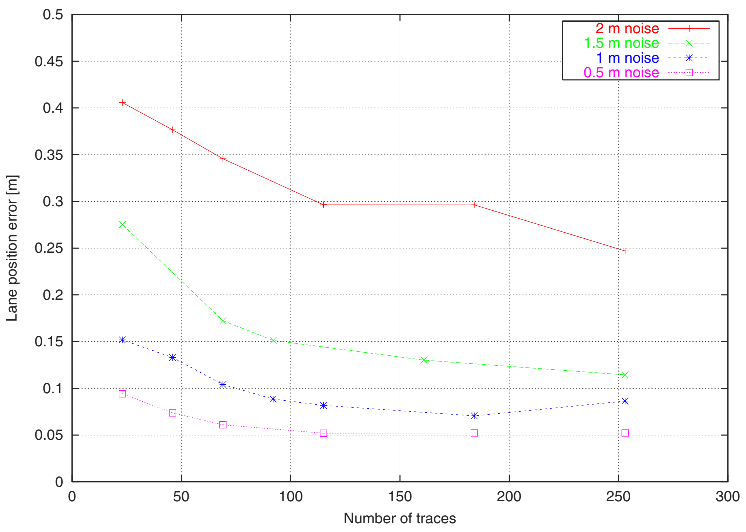

Figure 12. Average error in lane offsets

$C \leftarrow max_{(u,v)\in E}\{w(u,v)\}$. Two-Level Buckets have top level and bottom level of length $\lceil\sqrt{C+1}\rceil+1$, yielding the run time $O(e+n\sqrt{C})$. An implementation with *radix heaps* uses buckets of sizes $1,1,2,4,8,\ldots$ and imply $O(e+n\log C)$ run time, two-level heap improve the bound to $O(e+n\log C/\log\log C)$ and a hybrid with Fibonacci heaps yields $O(e+n\sqrt{\log C})$ [1]. This algorithm is almost linear in practice, since when assuming 32 bit integers we have $\lceil\sqrt{\log C}\rceil \leq 6$. The currently best result are *component trees:* with $O(n+e)$ time for undirected SSSP on a random access machine of word length w with integer edge weights in $[0..2^w-1]$ [31]. However, the algorithm is quite involved and likely not to be practical.

4.2 Planar Graphs

Travel graphs have many additional features. First of all, the number of edges is likely to be small. In the trail graph the number of edges equals the number of nodes minus 1, and for l trails $T_1,\ldots,T_l$ we have $|T_1|,\ldots,|T_l|-l$ edges in total. By introducing k intersections the number of edges increases by $2k$ only. Even if intersections coincide travel graphs are still planar, and by Eulers formula the number of edges is bounded by at most three times the number of nodes. Recall, that for the case of planar graphs linear time algorithms base on graph separators and directly lead to network flow algorithms of the same complexity [17].

If some intersections were rejected by the algorithms to allow non-intersecting crossing like bridges and tunnels, the graph would loose some of its graph theoretical properties. In difference to general graphs, however, we can mark omitted crossings to improve run time and storage overhead.

```
Dijkstra:                                                          A*

$Open \leftarrow \{(s,0)\}$                                        $Open \leftarrow \{(s,h(s))\}$
$Closed \leftarrow \{\}$
while ($Open \neq \emptyset$)
  $u \leftarrow Deletemin(Open)$
  $Insert(Closed,u)$
  if ($goal(u)$) return $u$
  for all $v$ in $\Gamma(u)$
    $f'(v) \leftarrow f(u) + w(u,v)$                               $+h(v) - h(u)$
    if ($Search(Open,v)$)
      if ($f'(v) < f(v)$)
        $DecreaseKey(Open(v,f'(v))$
    else if not ($Search(Closed,v)$)
      $Insert(Open,(v,f'(v))$
```

Table 1. Implementation of Dijkstra's SSSP algorithm vs. A*.

4.3 Frontier Search

Frontier search [18] contributes to the observation that the newly generated nodes in any graph search algorithm form a connected horizon to the set of expanded nodes, which is omitted to save memory.

The technique refers to Hirschberg's linear space divide-and-conquer algorithm for computing maximal common sequences [16]. In other words, frontier search reduces a $(d+1)$-dimensional memorization problem into a d-dimensional one. It divides into three phases. In the first phase, a goal t with optimal cost f^* is searched. In the second phase the search is re-invoked with bound $f^*/2$; and by maintaining shortest paths to the resulting fringe the intermediate state i from s to t is detected. In the last phase the algorithm is recursively called for the two subproblems from s to i, and from i to t.

4.4 Heuristic Search

Heuristic search includes an additional node evaluation function h into the search. The estimate h, also called heuristic, approximates the shortest path distance from the current node to one of the goal nodes. A heuristic is *admissible* if it provides a lower bound to the shortest path distance and it is *consistent*, if $w(u,v) + h(v) - h(u) \geq 0$. Consistent estimates are admissible.

Table 1 also shows the small changes in the implementation of A* for consistent estimates with respect to Dijkstra's SSSP algorithm. In the priority queue *Open* of generated and not expanded nodes, the f-values are tentative, while in set *Closed* the f-values are settled. On every path from to the initial state to a goal node the accumulated heuristic values telescope, and if any goal node has estimate zero, the f values of each encountered goal node in both algorithms are the same. Since in Dijkstra's SSSP al-

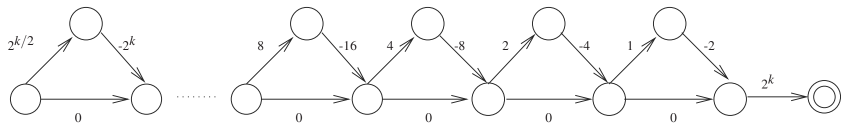

Figure 13. A graph with an exponential number of re-openings

gorithm the f-value of all expanded nodes match their graph theoretical shortest path value we conclude that for consistent estimates, A* is complete and optimal.

Optimal solving the SSSP problem for admissible estimates and negative values of $w(u,v)+h(v)-h(u)$ leads to re-openings of nodes: already expanded nodes in *Closed* are pushed back into the search frontier *Open*. If we consider $w(u,v)+h(v)-h(u)$ as the new edge costs, Fig. 13 gives an example for such a re-weighted graphs that leads to exponentially many re-openings. The second last node is re-opened for every path with weight $\{1,2,\ldots,2^k-1\}$. Recall that if h is consistent, no reopening will be necessary at all.

In route planning the Euclidean distance of two nodes is a heuristic estimate defined as $h(u) = \min_{g \in G} ||g-u||_2$ for the set of goal nodes G is both admissible and consistent Admissibility is granted, since no path on any road map can be shorter than the flight distance, while consistency follows from the triangle inequality for shortest path. For edge $e=(u,v)$ we have $\min_{g \in G} ||g-v||_2 \leq \min_{g \in G} ||g-u||_2 + w(u,v)$. Since nodes closer to the goal are more attractive, A* is likely to find the optimum faster. Another benefit from this point of view is that all above advanced data structures for node maintenance in the priority as well as space saving strategies like frontier search can be applied to A*.

5 Related Work

Routing schemes often run on external maps and external maps call for refined memory maintenance. Recall that external algorithms are ranked according to *sorting complexity* $O(sort(n))$, i.e., the number of external block accesses (I/Os) necessary to sort n numbers, and according to *scanning complexity* $O(scan(n))$, i.e., the number of I/Os to read N numbers. The usual assumption is that N is much larger than B, the block size. Scanning complexity equals $O(n/B)$ in a single disk model. On planar graphs, SSSP runs in $O(sort(n))$ I/Os, where n is the number of vertices. As for the internal case the algorithms apply graph separation techniques [30]. For general BFS at most $O(\sqrt{n \cdot scan(n+e)} + sort(n+e))$ I/Os [20] are needed, where e is the number of edges in the graph. Currently there is no $o(n)$ algorithm for external SSSP. On the other hand, $O(n)$ I/Os are by far too much in route planning practice.

Fortunately, one can utilize the spatial structure of a map to guide the secondary mapping strategy with respect to the graph embedding. The work of [11] provides the new search algorithm and suitable data structures in order to minimize page faults by a local reordering of the sequence of expansions. Algorithm *Localized A** introduces an operation *deleteSome* instead of strict *deleteMin* into the A* algorithm. Nodes corresponding to an active page are preferred. When maintaining an bound α on obtained solution lengths until *Open* becomes empty the algorithm can be shown to be complete and optimal. The loop invariant is that there is always a node on the optimal solution path with correctly estimated accumulated cost. The authors prove the correctness and completeness of the approach and evaluate it in a real-world scenario of searching a large road map in a commercial route planning system.

In many fields of application, shortest path finding problems in very large graphs arise. Scenarios where large numbers of on-line queries for shortest paths have to be processed in real-time appear for example in traffic information systems. In such systems, the techniques considered to speed up the shortest path computation are usually based on pre-computed information. One approach proposed often in this context is a space reduction, where pre-computed shortest paths are replaced by single edges with weight equal to the length of the corresponding shortest path. The work of [29] gives a first systematic experimental study of such a space reduction approach. The authors introduce the concept of multi-level graph decomposition. For one specific 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.

In the scenario of a central information server in the realm of public railroad transport on wide area networks a system has to process a large number of on-line queries for optimal travel connections in real time. The pilot study of [28] focuses on travel time as the only optimization criterion, in which various speed-up techniques for Dijkstra's algorithm were analyzed empirically.

Speed-up techniques that exploit given node coordinates have proven useful for shortest-path computations in transportation networks and geographic information systems. To facilitate the use of such techniques when coordinates are missing from some, or even all, of the nodes in a network [4] generate artificial coordinates using methods from graph drawing. Experiments on a large set of train timetables indicate that the speed-up achieved with coordinates from network drawings is close to that achieved with the actual coordinates.

6 Conclusions

We have seen a large spectrum of efficient algorithms to tackle different aspects of the route planning problem based on a given set of global positioning traces.

For trail graph inference the algorithm of Bentley and Ottmann has been modified and shown to be almost as efficient as the fastest shortest path algorithms. Even though this solves the basic route planning problem, different enhanced aspects are still open. We indicate low memory consumption, localized internal computation, and fast on-line performance as the most challenging ones.

Map inference and map matching up to lane accuracy suite better as a human-computer interface, but the algorithmic questions include many statistical operations and are non-trivial for perfect control. On the other hand, map inference based on GPS information saves much money especially to structure unknown and continuously changing terrains.

Low-end devices will improve GPS accuracy especially by using additional inertia information of the moving object, supplied e.g. by a tachometer, an altimeter, or a compass. For (3D) map generation and navigation other sensor data like sonar and laser (scans) can be combined with GPS e.g. for outdoor navigation of autonomous robots and in order to close uncaught loops.

The controversery if the GPS routing problem is more a geometrical one (in which case the algorithm of Bentley/Ottmann applies) or a statistical one (in which clustering algorithms are needed) is still open. At the moment we expect statistical methods to yield better and faster results due to their data reduction and refinement aspects and we expect that a geometrical approach will not suffice to appropriately deal with a large and especially noisy data set. We have already seen over-fitting anomalies in the statistic analyses. Nevertheless, a lot more research is needed to clarify the the quest of a proper static analysis of GPS data, which in turn will have a large impact in the design and efficiency of the search algorithms.

We expect that in the near future, the combination of positioning and precision map technology will give rise to a range of new vehicle safety and convenience applications, ranging from warning, advice, up to automated control.

Acknowledgments S. Edelkamp is supported by DFG in the projects *heuristic search*, and *directed model checking*. Thanks to Shahid Jabbar for visualizing the GPS trace.

References

1. R. K. Ahuja, K. Mehlhorn, J. B. Orbin, and R. E. Tarjan. Faster algorithms for the shortest path problem. *Journal of the ACM*, pages 213–223, 1990.
2. I. J. Balaban. An optimal algorithm for finding segment intersection. In *ACM Symposium on Computational Geometry*, pages 339–364, 1995.
3. J. L. Bentley and T. A. Ottmann. Algorithms for reporting and counting geometric intersections. *Transactions on Computing*, 28:643–647, 1979.
4. U. Brandes, F. Schulz, D. Wagner, and T. Willhalm. Travel planning with self-made maps. In *Workshop on Algorithm Engineering and Experiments (ALENEX)*, 2001.
5. B. Chazelle. Reporting and counting segment intersections. *Computing System Science*, 32:200–212, 1986.
6. B. Chazelle and H. Edelsbrunner. An optimal algorithm for intersecting lines in the plane. *Journal of the ACM*, 39:1–54, 1992.
7. T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. The MIT Press, 1990.
8. R. B. Dial. Shortest-path forest with topological ordering. *Communication of the ACM*, 12(11):632–633, 1969.
9. E. W. Dijkstra. A note on two problems in connection with graphs. *Numerische Mathematik*, 1:269–271, 1959.
10. P. Doucette, P. Agouris, A. Stefanidis, and M. Musavi. Self-organized clustering for road extraction in classified imagery. *Journal of Photogrammetry and Remote Sensing*, 55(5-6):347–358, March 2001.
11. S. Edelkamp and S. Schroedl. Localizing A*. In *National Conference on Artificial Intelligence (AAAI)*, pages 885–890, 2000.
12. M. L. Fredman and R. E. Tarjan. Fibonacci heaps and their uses in improved network optimization algorithm. *Journal of the ACM*, 34(3):596–615, 1987.

13. S. Handley, P. Langley, and F. Rauscher. Learning ot predict the duration of an automobile trip. In *Knowledge Discovery and Data Mining (KDD)*, pages 219–223, 1998.
14. A. C. Harvey. *Forecasting, Structural Time Series Models and the Kalman Filter*. Cambridge University Press, 1990.
15. C. A. Hipke. *Verteilte Visualisierung von Geometrischen Algorithmen*. PhD thesis, University of Freiburg, 2000.
16. D. S. Hirschberg. A linear space algorithm for computing common subsequences. *Communications of the ACM*, 18(6):341–343, 1975.
17. P. Klein, S. Rao, M. Rauch, and S. Subramanian. Faster shortest-path algorithms for planar graphs. *Special Issue of Journal of Computer and System Sciences on selected papers of STOC 1994*, 55(1):3–23, 1997.
18. R. E. Korf and W. Zhang. Divide-and-conquer frontier search applied to optimal sequence allignment. In *National Conference on Artificial Intelligence (AAAI)*, pages 910–916, 2000.
19. J. B. MacQueen. Some methods for classification and analysis of multivariate observations. In *Symposium on Math, Statistics, and Probability*, volume 1, pages 281–297, 1967.
20. K. Mehlhorn and U. Meyer. External-memory breadth-first search with sublinear I/O. In *European Symposium on Algorithms (ESA)*, 2002.
21. G. W. Milligan and M. C. Cooper. An examination of procedures for determining the number of clusters in a data set. *Psychometrika*, 50(1):159–179, 1985.
22. C. Navigation Technologies, Sunnyvale. Software developer's toolkit, 5.7.4 solaris edition, 1996.
23. B. W. Parkinson, J. J. Spilker, P. Axelrad, and P. Enge. *Global Positioning System: Theory and Applications*. American Institute of Aeronautics and Astronautics, 1996.
24. L. Piegl and W. Tiller. *The nurbs book*. Springer, 1997.
25. C. A. Pribe and S. O. Rogers. Learning to associate driver behavior with traffic controls. In *Proceedings of the 78th Annual Meeting of the Transportation Review Board*, Washington, DC, January 1999.
26. S. Rogers, P. Langley, and C. Wilson. Mining GPS data to augment road models. In *Knowledge Discovery and Data Mining (KDD)*, pages 104–113, 1999.
27. S. Schroedl, S. Rogers, and C. Wilson. Map refinement from GPS traces. Technical Report RTC 6/2000, DaimlerChrysler Research and Technology North America, Palo Alto, CA, 2000.
28. F. Schulz, D. Wagner, and K. Weihe. Dijkstra's algorithm on-line: An empirical case study from public railroad transport. *Journal of Experimental Algorithmics*, 5(12), 2000.
29. F. Schulz, D. Wagner, and C. Zaroliagis. Using multi-level graphs for timetable information. In *Workshop on Algorithm Engineering and Experiments (ALENEX)*, 2002.
30. L. Thoma and N. Zeh. I/O-efficient algorithms for sparse graphs. In *Memory Hierarchies*, Lecture Notes in Computer Science. Springer, 2002. To appear.
31. M. Thorup. Undirected single-source shortest paths with positive integer weights in linear time. *Journal of the ACM*, 46:362–394, 1999.
32. J. Wang, S. Rogers, C. Wilson, and S. Schroedl. Evaluation of a blended DGPS/DR system for precision map refinement. In *Proceeedings of the ION Technical Meeting 2001*, Institute of Navigation, Long Beach, CA, 2001.