Predicting Route Targets Based on Optimality Considerations

Jörg Roth
Department of Computer Science
Nuremberg Institute of Technology
Nuremberg, Germany
Joerg.Roth@th-nuernberg.de

***Abstract*—In this paper we present an approach to predict a target of a mobile user on the move. After observing the movement from a starting point, we are able to create possible extrapolations of routes. Our basic assumption: a mobile user tries to move efficiently, thus only a certain set of destinations is reasonable. We use a road network that contains information about movement costs to detect reasonable movements, but we do not expect theoretical optimal paths. We are able to model different efficiency goals and different degrees of optimality. We present an efficient algorithm to actually compute the set of reasonable targets that avoids brute force computation. In contrast to existing work to predict route destinations, we do not require a learning phase to collect an archive of former routes.**

Keywords—route planning, target prediction, road maps

I. INTRODUCTION

Current mobile users often carry mobile devices that are able to detect their current position, often based on satellite navigation systems. A mobile device can thus easily store recent locations for a certain time. In this paper we introduce an approach to predict all possible targets based on the current route, i.e. the positions from the last starting point to the current position. The knowledge about possible targets can be useful for several services and applications:

- A community service can proactively register a user to potential targets. Community members can be informed *before* a user arrives, e.g. "Bob will approach downtown in 10 minutes".
- Information services can provide information for people on the move. They can, e.g., warn drivers who are going to drive into a restricted low-emission zone, can suggest free parking places in the target area or inform about traffic jams.
- An advanced tourist guide can suggest interesting sites in the current walking direction. Similar services can be useful for, e.g. shopping (detect shops) or hiking (detect accommodations).

Predicting targets has a long tradition. The common approach is to learn the user's habits from former trips. The drawbacks: we have to undergo a learning phase and it completely fails, if the user moves to a certain destination for the first time.

978-1-4799-5350-9/14/$31.00 ©2014 IEEE

In this paper we introduce a novel approach: we only process the positions of the current trip and extrapolate the route using a road network. We assume a mobile user tries to hold a certain degree of optimality when moving across the road network. We can use this assumption to predict reasonable targets. Our approach contains two parts:

- We first define a measure of *target orientation*. This is due to the fact that users may not necessarily drive on theoretically optimal routes.
- We create a mechanism that extends an existing route to all targets that hold a given measure of target orientation.

This approach is not able to detect the single actual destination of a trip; instead it computes a superset of all reasonable targets. We later describe it as a polygonal area that encloses all targets. If the mobile user drives reasonably (described by our measure of target orientation), the actual destination will be part of our computed target region.

We are free to define any measure of target orientation. But it turned out that most of them only enable brute force mechanisms to compute a target area. Thus, we have to carefully select a measure that a) reflects the intention of target oriented movement, and b) supports an efficient target area computation. We later introduce such a measure.

II. RELATED WORK

Several approaches deal with the problem of route and target prediction. The majority of existing work tries to learn important routes from the past. For this, a mobile user is observed for a longer time (e.g. some weeks) to create an archive of routes. These approaches usually answer two questions: a) How can an automatic process extract routes from stored motion logs? b) Once started driving, how can we find a similar leading path in our route archive? Both answers usually base on a similarity function that compares route geometries – existing work (e.g. [2], [5], [7], [9]) may differ in the respective similarity function. The measure of similarity can be improved with additional mechanisms: [6] applies a Hidden Markov Model to evaluate a sequence of route segments. [3] not only creates a history of routes, but tries to organize them as a route tree. [4] additionally considers day and daytime to evaluate

similarity of routes. Once we predicted a probable route, we can also use this information to improve the positioning system [10].

Another group of approaches not want to predict routes but try to identify important targets ([12], [13]). These targets can be computed with the help of geometric clustering without to know the actual road network.

A solution to a smaller problem is presented in [11]: instead of finding out the final destination, the approach tries to predict only the next road segment, i.e. the turn at the *next* crossing. This information may be useful for routing applications, but is too specific for more complex services.

The approach described in [8] is the closest to our work: it computes possible target areas assuming a driver mainly uses efficient routes. For this, the road network is split into cells of 1 km x 1 km. Between each pair of cells a route planning algorithm computes the shortest path between road points nearest to the cell centres. A driven route can be considered as a list of traversed cells. As all shortest paths between cells centres are known, possible target cells can thus be identified. The major drawback is the cell-based structure: we cannot identify targets smaller than cells and the road network is simplified to few representatives – the cell centres. For small routes and dense road networks, this approach is not applicable as it actually based on a brute-force approach. It cannot be transferred to smaller structures than cells, because the number of routing permutations would get critical.

III. Predicting Target Regions

A. The General Idea

The idea is based on an assumption: a mobile user tries to move efficiently across a road network. The degree of efficiency depends on the respective user, but moving from *A* to *B* people intuitively try to reduce costs – whatever costs mean. In the following we use the terms *'driving'* and *'driver'*, but this approach can also be transferred to pedestrians or bicycle riders, i.e. all types of movement that can be modelled by a road network with a costs function.

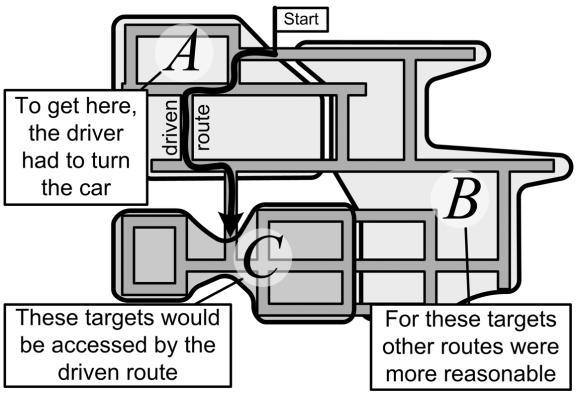


Fig. 1. General idea to identify the target area

Fig. 1 presents the idea. We observe a driven route. At a certain point we try to predict the targets. We can argue as follows:

- Area *A* would require the driver to turn and drive back, thus this area unlikely covers the target.
- For area *B*, the driver would have chosen another route, which, e.g., directly started in the opposite direction.
- Area *C* contains targets that would extend the existing route in a reasonable way. We assume this area contains the actual target.

We can thus identify two characteristics of the target: a) reasonable route extensions do not contain any *back-driving* and b) targets cannot be reached more efficiently by *alternative routes*. We later express these properties as *locally target oriented* (no back-driving) and *globally target oriented* (no alternative routes).

B. Preliminaries and Definitions

We assume a mechanism to identify a start of a movement. A simple approach e.g. would monitor the current position and separate halts (position remains nearly constants) from movement (position significantly changes). We further assume a mechanism that frequently stores the current position – not necessarily with constant time intervals. Let $P=\{p_1,\ldots p_n\}$ denote the set of stored positions. The positions are timely ordered by their index, thus we can define an ordering $\angle$: $p_i \angle p_j$ if $i<j$, i.e. p_i is measured before p_j.

Essential for our approach is a graph model of the road network consisting of nodes (crossings) and edges (road segments between crossings). To each edge we assign driving costs – positive numbers that indicate a relevant measure that a driver wants to minimize, e.g. driving distance, driving time or fuel consumption. Costs can be computed beforehand from the road geometry (including inclination), road type (e.g. motorway) and further properties (e.g. speed limit, pavement). We presented an approach to compute a road network including costs in [14], [15].

Let Q denote all crossings of the road network and $c(q_i, q_j)$ the driving costs between two connected crossings $q_i, q_j \in Q$. Further definitions:

- $c^*(a, b)$ are the *minimal costs* (i.e. on the optimal route) between a and b, where $a, b \in P \cup Q$.
- $k(p_a, p_b) = \sum_{i=a}^{b-1} c^*(p_i, p_{i+1})$ are *real costs* of the driven route between $p_a, p_b \in P, p_a \angle p_b$.

Q may contain several millions of crossings (approx. 11 million for Germany). c^* can be computed using shortest path algorithms (foremost A* or variations of A*). Important: original routing algorithms such as A* minimize costs for routes from *crossing to crossing*. Real routes, however, usually start and terminate *between* crossings. There exist different solutions to compute such routes. One is to include a virtual start and target crossing. This sometimes is technically difficult as the road network usually is stored in an optimized manner (e.g. in compressed files) that cannot easily be changed at runtime. Thus, we use more sophisticated approaches, such as described

in [15]. Independently from the actual choice, we assume that c^* may take any point in the road network, not only crossings.

Looking at the driven route, k is an approximation: we assume that a driver *optimally* drives between two measured positions. If succeeding measurements are close enough, this assumption obviously is useful.

Some properties: for $p_a, p_b, p_c \in P$, $p_a \angle p_b \angle p_c$; $q_i, q_j, q_k \in Q$:

$$k(p_a, p_b) \geq c^*(p_a, p_b) \quad (1)$$

$$k(p_a, p_c) = k(p_a, p_b) + k(p_b, p_c) \quad (2)$$

$$c^*(q_i, q_k) \leq c^*(q_i, q_j) + c^*(q_j, q_k) \quad (3)$$

$$c^*(p_a, q_i) \leq k(p_a, p_b) + c^*(p_b, q_i) \quad (4)$$

$$\text{if } c^*(p_a, q_i) = k(p_a, p_b) + c^*(p_b, q_i) \text{ then } p_b \text{ is part of the optimal route from } p_a \text{ to } q_i \quad (5)$$

(3) is also known as *triangle inequality* of road networks. (4) follows from (1)-(3). The right side in (5) is obviously a segmentation of the optimal route, thus $k(p_a, p_b)$ represents the optimal path from p_a to p_b.

C. Measuring the Target Orientation

In the following we model the degree of efficiency, called the *target orientation*. Even though we are in principle free to create any model that can be mathematically formulated, only few formulas (especially ours) provide an efficient approach to predict extensions. We introduce for $p_a, p_b \in P$, $p_a \angle p_b$

$$t(p_a, p_b) = \frac{c^*(p_a, p_b)}{k(p_a, p_b)} \quad (6)$$

that represents the *degree of target orientation* for a part of the driven route. We further define $t(p_a, p_a)=1$ and for $p_b \in P$, $dist>0$

$$t_d(dist, p_b) = t(p_a, p_b) \text{ where } a = MIN\{i \mid k(p_i, p_b) \leq dist\} \quad (7)$$

Some properties: $0 \leq t(p_a, p_b) \leq 1$, $t_d(\infty, p_a) = t(1, p_a)$, $t_d(0, p_a) = 1$. Values of t can be interpreted as follows: $t(p_a, p_b) \approx 1$: optimal or nearly optimal route, $t(p_a, p_b)=0.5$: driver needs twice as much as optimal route and $t(p_a, p_b) \approx 0$: route was a roundtrip. Thus, t, t_d provide appropriate measures for how target oriented the route was. To reflect the two different requirements for targets (as described in section A), we introduce a *global* and *local* target orientation property for route point p_a:

$$t_{global}(p_a) = t_d(\infty, p_a) \quad (8)$$

$$t_{local}(p_a) = t_d(c_{local}, p_a) \quad (9)$$

t_{global} relates the optimal route to the driven route and detects alternative routes. t_{local} indicates the target orientation for the 'last mile' of the route and detects back-driving. c_{local} specifies the distance, where a difference from the optimal route is a result of wrong driving and not a result of a reasonable alternative route. It is expressed in the respective cost model. Useful values are e.g. 60 seconds (if costs are driving times) or 600 meters (if costs are distances).

We cannot expect a driver to always drive theoretically optimal routes, because a) people's brains do not execute optimal route planning algorithms and more important b) the underlying cost model does not necessarily reflect the real world. People often know better routes than route planning tools. To consider alternative routes with slightly higher costs, we use the following definition of *target orientation*:

- a route point p_a is *target oriented*, if $t_{local}(p_a) \geq \tau_{local}$ and $t_{global}(p_a) \geq \tau_{global}$;
- a *route is target oriented*, if all its route points are target oriented.

Typical values are $\tau_{global}=0.8$, $\tau_{local}=0.7$. We are now able to model a driver's intention by the tuple $(c, c_{local}, \tau_{global}, \tau_{local})$.

D. The g Array

Our goal is to find a mechanism to extend a driven route that holds a certain degree of target orientation without the use of brute-force algorithms. More formally: given a driven route P that is target oriented according to $(c, c_{local}, \tau_{global}, \tau_{local})$; compute all targets q_i that extend the driven route to a target oriented route.

A brute force algorithm would iterate through all nodes q_i in the road network and measure the target orientation. But as t requires the costly computation of c^* (with computation times of several 100 ms for longer routes), this approach is not reasonable for millions of crossings.

Our approach is based on g arrays (named after the cost array in A* [18]). They store for a certain start p_a and crossing q_i the minimal costs from start to q_i. More formally:

$$g_a[i] = \begin{cases} c^*(p_a, q_i) & \text{if } q_i \text{ fulfils required conditions} \\ -1 & \text{otherwise} \end{cases} \quad (10)$$

The array may only partly be filled. We use a Dijkstra approach that can also be considered as A* without a target and the trivial estimation ([18], [19]). We may define an upper bound c_{max} for g values, i.e. only target crossings are considered that can be reached within a certain cost limit. In addition we can define a *condition* (see below). If a crossing later has a non-negative g entry, this condition is true for the crossing and for all crossings on the optimal path from p_a to this crossing.

The idea of this algorithm is to take the next non-closed crossing with the lowest current distance from the start. This crossing is then *expanded*: we check if its neighbours use this crossing as last hop. The most expensive operation is 'identify the q_i with minimal $g_a[q_i]$'. If we use an ordered list to implement the set *open*, we can reduce the lookup costs for the next crossing to $\log(|open|)$.

We can sketch the algorithm to create g arrays as follows:

compute_g(a, c_{max}, *condition*)

```
closed←{}; open←{p_a};
g_a[a]←0; g_a[q_i]← -1 for all q_i≠p_a;
do {
    identify q_i∈open with minimal g_a[q_i];
    open←open\{q_i}; closed←closed ∪ {q_i};
    if not condition { g_a[q_i]← -1; continue; }
    for all neighbours q_j of q_i {          // expand crossing
        if q_j ∉ closed {
            g_new=g_a[q_i]+c(q_i, q_j);
            if g_new≤c_max and (q_j ∉ open or g_new<g_a[q_j]) {
                open←open ∪ {q_j};
                g_a[q_j]←g_new;
                memorize link q_i→q_j if required // *
            }
        }
    }
} while open ≠ {};
return g_a;
```

For better clearness, we assume $p_a \in Q$. As stated above, this is not necessarily true, but the respective solutions (e.g. insert p_a as a *virtual* crossing) would overload the code above. In addition, efficient implementations model *not visited*, *open*, *closed* as states attached to a crossing, thus, element-of checks and set operations are executed in constant time.

Note that sometimes we do not only want to know the costs to a certain crossing but also the optimal route to get there. If required, we can store the last link to each crossing (* in the pseudo code above), thus can easily collect the entire route.

E. Extrapolation of Target Oriented Routes into the Future

The goal is to extend a driven route that holds a certain degree of target orientation to all possible routes that also hold this degree of target orientation. The endpoints of these routes form the *target region*. We start with an observation. Consider, we compute

```
g_1←compute_g(1, ∞, true);
g_n←compute_g(n, ∞, true);
T_naive←{q_i | g_1[q_i] – g_n[q_i]=c*(p_1, p_n) };
```

According to property (5), all crossings inside T_{naive} are *optimal* extensions of the route p_1 to p_n. Thus, this algorithm provides a first version to compute a target region. The great benefit: we avoid brute force as we mainly have to subtract two arrays. However, we have some disadvantages:

- The g arrays of p_1 and p_n have to be fully generated.
- Only optimal routes (according to c) are considered.
- Only the first and last route points are considered, not inner points of the driven route.

Even though T_{naive} does not fully reflect the driver's intention, it forms the basis for our approach. The improvement of this naïve approach: rather than only collection fully optimal routes, we use our notion of target orientation that takes into account the entire driven route. For this, we first define reasonable extensions of t, t_d, t_{global} and t_{local}:

$$t_{ex}(p_a, p_b, q_i) = \frac{c^*(p_a, q_i)}{k(p_a, p_b) + c^*(p_b, q_i)} \tag{11}$$

$$t_{exd}(dist, p_b, q_i) = t_{ex}(p_a, p_b, q_i) \text{ where}$$
$$a = MIN\{j \,|\, k(p_j, p_b) + c^*(p_b, q_i) \le dist\} \tag{12}$$

$$t_{exglobal}(p_a, q_i) = t_{exd}(\infty, p_a, q_i) \tag{13}$$

$$t_{exlocal}(p_a, q_i) = t_{exd}(c_{local}, p_a, q_i) \tag{14}$$

The idea behind t_{ex} is similar to t: it relates optimal routes to driven routes, but it does not only consider routes from p_a to p_b, but also their possible extensions to a crossing q_i. The measured route now consists of two parts: the driven route p_a to p_b and the possible extension p_b to q_i. As the extension is unknown and we want to know whether the driver *in principle* can reach this crossing in a target oriented manner, we assume the optimal route p_b to q_i. As a result, t_{ex} and t_{exd} are reasonable extensions of t and t_d respectively. We now can define the target region:

$$T_{globloc} = \{q_i \mid \; t_{exglobal}(p_n, q_i) \ge \tau_{global} \wedge \; t_{exlocal}(p_n, q_i) \ge \tau_{local}\} \tag{15}$$

We sketch the code for $T_{globloc}$ as follows.

```
find_dist(d, b)  // Find the first route point that
                 // has distance up to d to route point p_b
  j←b; m←b;
  while (j≥1 and k(p_j, p_b)≤d) { m←j;  j←j–1; }
  return m;

t_exglobal(a, q_i)        // Compute t_exglobal
  return g_1[q_i]/(k(p_1, p_a)+g_a[q_i]);

t_exlocal(a, q_i)         // Compute t_exlocal
  j←find_dist(c_local – g_a[q_i], a);
  return g_j[q_i]/(k(p_j, p_a)+g_a[q_i]);

T_globloc()                          // Compute T_globloc
  g_1←compute_g(1, c_max, true);     // required for t_exglobal
  j_min←max(2, find_dist(c_local, n)); // first array for t_exlocal
  for j from j_min to n–1            // further arrays for t_exlocal
    g_j←compute_g(j, ∞, g_1[q_i]≥0);
  g_n←compute_g(n, ∞, g_1[q_i]≥0 and
                      t_exglobal(n, q_i)≥τ_global and
                      t_exlocal(n, q_i)≥τ_local);
  return all q_i with g_n[q_i]≥0;
```

The pseudo code for **find_dist** only provides the idea and is simplified for better clearness. An efficient implementation would use a binary search approach to find the appropriate route point j. Another optimization: k (and thus c^*) are computed very often for the same values. It is reasonable to precompute and store $k(p_a, p_b)$ once for every pair p_a, p_b of route points $p_a \angle p_b$.

The developer can set an appropriate value of c_{max} that defines the maximum route length for which target areas are

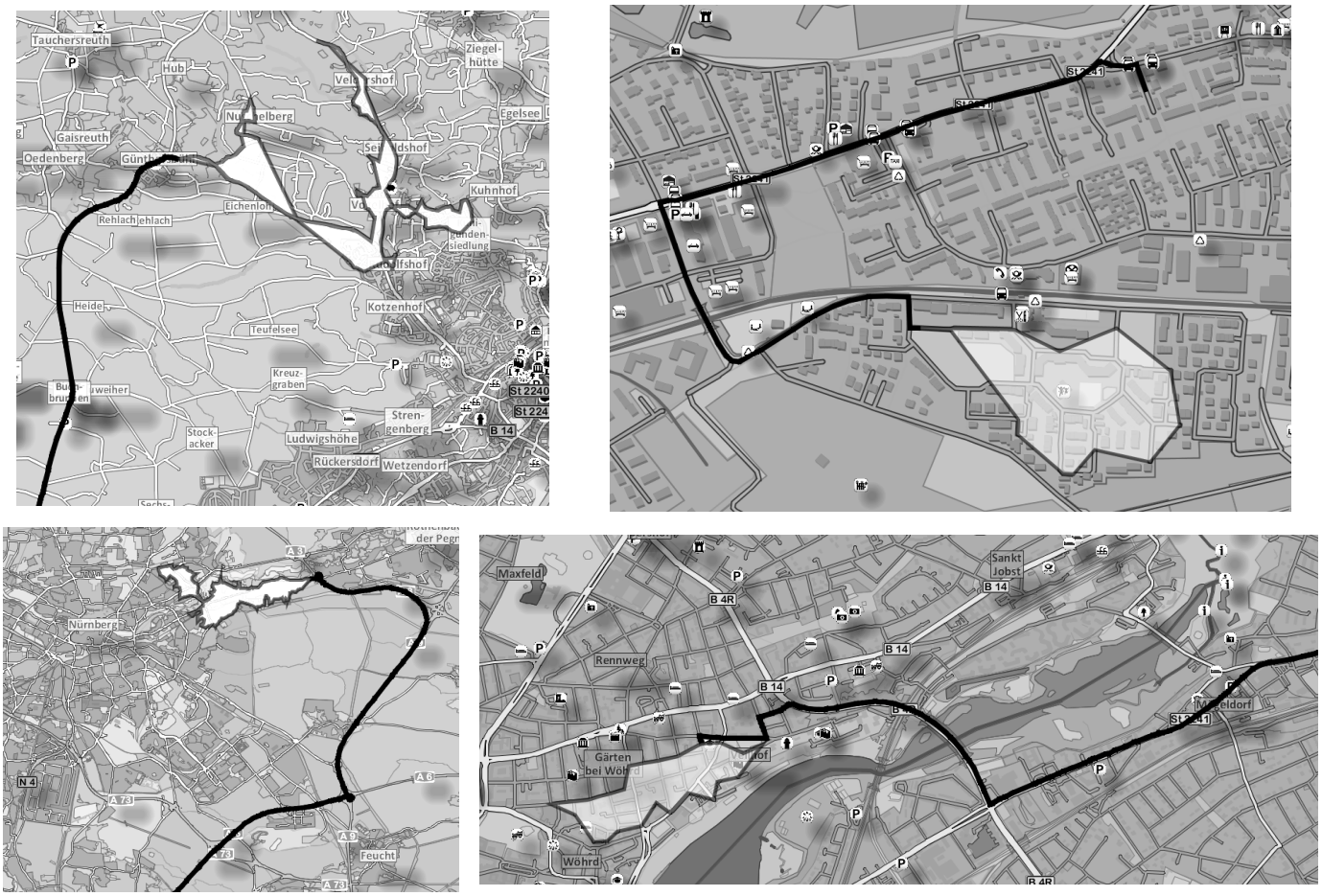


Fig. 2. Example target regions (τ_{global}=0.8, τ_{local} =0.7, c_{local}=60 s, c_{max}=1 h)

computed. We process the value during the generation of g_1. Further g arrays do not go beyond c_{max} because of the condition $g_1[q_i] \geq 0$.

Fig. 2 shows target regions of driven routes. Actually, the target region is a set of crossings. For better clearness, we painted the concave hull of these crossings.

F. Improving Execution Time

In the algorithm above, the array creation g_n only expands crossings, if they hold the global *and* local condition, as on a target oriented route *every* crossing must hold both conditions. Thus, we can significantly reduce the expansion cost for the last array, if we only expand crossings that actually can be part of a target oriented route.

We can significantly reduce the g array creation even more, if we consider a property of t_{ex}. For three route points p_a, p_{b1}, p_{b2} ($p_a \angle p_{b1} \angle p_{b2}$), crossing q_i: $t_{ex}(p_a, p_{b2}, q_i) \leq t_{ex}(p_a, p_{b1}, q_i)$, because

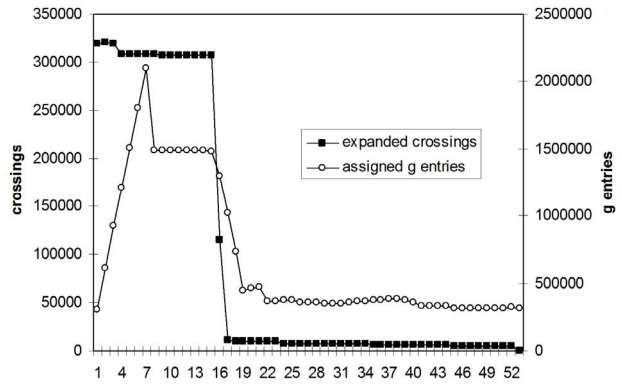
$$t_{ex}(p_a, p_{b2}, q_i) = \frac{c^*(p_a, q_i)}{k(p_a, p_{b2}) + c^*(p_{b2}, q_i)} \overset{(2)}{=} \frac{c^*(p_a, q_i)}{k(p_a, p_{b1}) + k(p_{b1}, p_{b2}) + c^*(p_{b2}, q_i)} \overset{(4)}{\leq} \frac{c^*(p_a, q_i)}{k(p_a, p_{b1}) + c^*(p_{b1}, q_i)} = t_{ex}(p_a, p_{b1}, q_i) \quad (16)$$

An important consequence of (16): for two route points p_{a1}, p_{a2} ($p_{a1} \angle p_{a2}$), crossing q_i: if $t_{exglobal}(p_{a1}, q_i) < \tau_{global}$ then also $t_{exglobal}(p_{a2}, q_i) < \tau_{global}$. Proof:

$$t_{exglobal}(p_{a1}, q_i) = t_{ex}(p_1, p_{a1}, q_i),$$
$$t_{exglobal}(p_{a2}, q_i) = t_{ex}(p_1, p_{a2}, q_i),$$
$$\text{because of (16) } t_{exglobal}(p_{a2}, q_i) \leq t_{exglobal}(p_{a1}, q_i),$$
$$\text{thus finally } t_{exglobal}(p_{a2}, q_i) < \tau_{global} \quad (17)$$

(17) is useful to limit the g array creation: if a certain node q_i is not globally target oriented for a route point p_{a1}, it also cannot be globally target oriented for a subsequent route point p_{a2}, thus we can exclude q_i for all further g arrays. Unfortunately, there is no such rule for the local target orientation. However, this property already significantly safes processing time.

Based on this consideration, we now can formulate an improved algorithm to compute $T_{globloc}$.



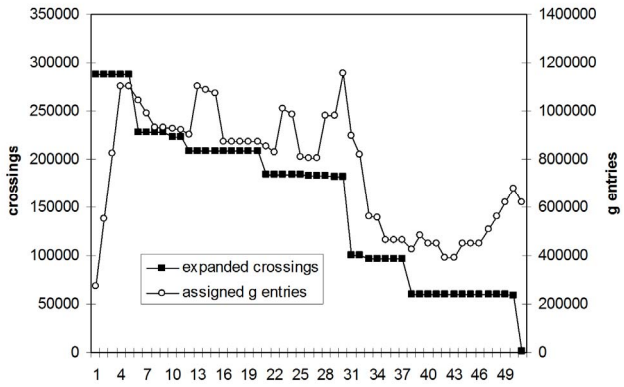


Fig. 3. Memory requirement and number of expansions

```
T_globloc()                          // Compute T_globloc
  g_1←compute_g(1, c_max, true);     // required for t_exglobal
  last←1;
  j_min←max(2, find_dist(c_local, n)); // first array for t_exlocal
  for j from j_min to n–1 {          // further arrays for t_exlocal
    g_j←compute_g(j, ∞, g_last[q_i]≥0 and
                        t_exglobal(j, q_i)≥τ_global); // (17)
    last←j;
  }
  g_n←compute_g(n, ∞, g_last[q_i]≥0 and
                      t_exglobal(n, q_i)≥τ_global and
                      t_exlocal(n, q_i)≥τ_local);
  return all q_i with g_n[q_i]≥0;
```

G. Consecutive Execution

Until now we strongly separated the collection of route points from target region computation. It is computed not before a driven route is completed to a certain point. Many usage scenarios, however, prefer an execution that consecutively processes route points one after another. E.g. consider an application that warns a driver whenever a potential target requires a road charge. As there is no certain route point to warn, the application wants to refine the target region for every new route point. We can easily modify the algorithm above to a consecutive variation:

```
process_globloc(n)
  if n=1
    g_1←compute_g(1, c_max, true);      // required for t_exglobal
  else {
    g_n←compute_g(n, ∞, g_n-1[q_i]≥0 and
                        t_exglobal(n, q_i)≥τ_global);   // (17)
    j_min ←max(2, find_dist(c_local, n));
    for m from 2 to j_min–1   // remove no longer used g arrays
        if g_m exists then remove g_m from memory;
  }

compute_globloc()
  return all q_i with g_n[q_i]≥0 and
                      t_exlocal(n, q_i)≥τ_local;
```

We use the code as follows:

- we call **process_globloc(*n*)** for every new route point p_n after it is measured;
- we call **compute_globloc()**, whenever we want to compute a target region based on the recently processed route points.

Inside **compute_globloc()** a real implementation would not iterate through all *g* array elements to collect target crossings. Instead, we would include this condition into **compute_g**. However, this improvement would overload the pseudo code above.

It depends on the actual application whether the consecutive variation is more suitable than the original version. The consecutive variation has to compute all *g* arrays – some of them may not be used later. On the other hand, a target region can be requested at any time without significant further computation.

Fig. 3 shows statistics of the consecutive executions of the two left routes in Fig. 2. We measured the number of *g* entries in memory that have an assignment (a non-negative value) and the number of newly expanded crossings. The values are presented for every route point, thus indicate the current memory and runtime requirement.

Note that the actual memory requirement for *g* entries can be higher, if we, e.g., model the *g* array as an actual array in program languages (e.g. int[] g) for runtime reasons. Then, we had to store a huge number of -1 values. In contrast, we may safe storage space (with the cost of longer runtime) and only store non-negative values in a hash table. It depends on the developer's choice, who should take into account the intended runtime environment, e.g. whether the algorithm should run on a smart phone device or server.

To give some impression about the memory requirements: we can store cost value, crossing state, link ID and crossing IDs in a packed structure of 10 bytes. The examples above require up to 2 million stored entries, i.e. 19 MByte of memory solely for the stored values. The organisation as a hash table requires additional memory (highly implementation-dependent thus difficult to determine). But as a result, the memory requirements for this algorithm cannot be considered as critical.

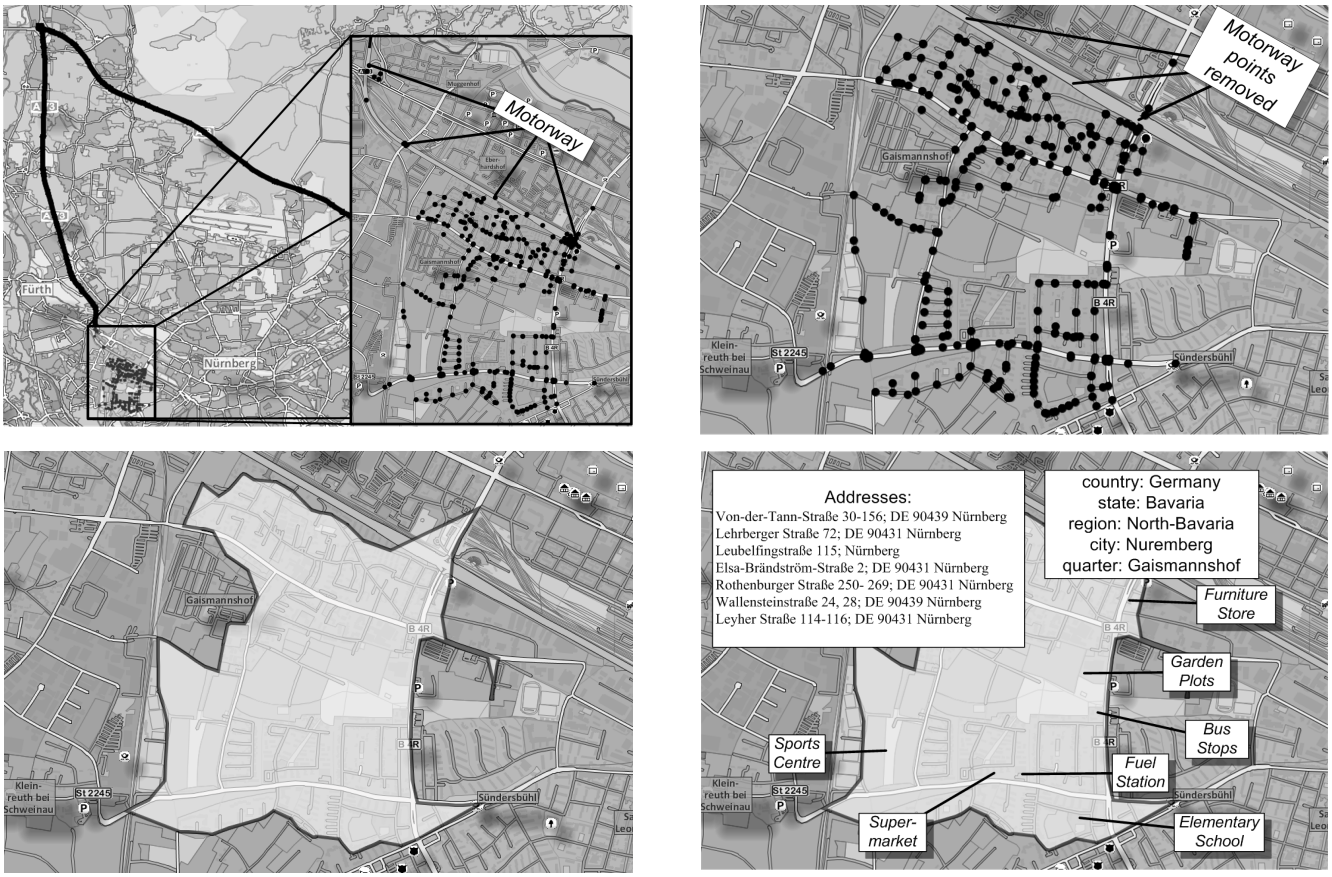


Fig. 4. Further processing steps of target regions

Looking at the runtime: most time-consuming are the **compute_g** calls. We measure the execution time on a typical PC (Intel Core i7-2600 CPU 3.4 GHz). It can execute 1.39 million crossing expansions per second. Based on the c_{max} above we require up to 0.32 million expansions per measured route point. Thus, execution time also is not critical.

In the charts, we can easily see a stepwise reduction of expanded crossing and stored *g* entries. Each reduction step is a result of passing a significant crossing that further confines the potential targets. If, e.g. the driver does *not* leave at a motorway exit, the locations connected to this exit are obviously not part of the target region. Due to the nature of our approach, such locations are then automatically removed from the next steps.

H. Further Processing of Target Areas

Until now, we consider a target region as the set of crossings (Fig. 4 top left). Regardless of the target, the *intention* of a trip is to stay somewhere or do something at the destination. Thus, the actual destination is usually not part of the road network, but somewhere in the nearer area. Further processing that reflects the actual intention is highly dependent on the actual service. However, here is a list of possible actions:

1. We remove road types from the result that usually are not target of a trip, but only passed over. Good examples are motorways: people usually do not drive to a certain motorway position as a final destination. We thus first remove all targets that belong to such road types (Fig. 4 top right).

2. We switch from a topological to a geometric representation, needed for further spatial queries (Fig. 4 bottom left). We perform the following steps: a) Collect waypoint coordinates of result links. b) Create a *concave hull* of these coordinates – a closed polygon that encloses a set of given positions. In contrast to the convex hull, there is no unique concave hull – usually a parameter specifies how much the polygon geometrically adapts to the given coordinates. We use an approach based on the Delaunay triangulation that requires $O(n \log n)$ steps [1]. Finally c) we may enlarge the polygon by some meters using the so-called *buffer* operation. This is because the actual target may be a neighbouring area of a road.

3. With the area polygon we can perform spatial queries on geo databases. We use our *HomeRun* platform [16], [17] that e.g. supports reverse geocoding requests (Fig. 4 bottom right). It answers for any given geometry: a) In which larger objects is this geometry embedded, e.g. country, state, city. b) The postal addresses enclosed by this geometry. c) Remarkable objects, e.g. shops, schools that are enclosed by the area geometry.

The types of relevant objects differ between applications. E.g. a driver support application may only be interested in free parking, whereas a tourist app may query touristic sight. A specific application or service can easily set specific object filters to only get interesting objects.

Further derived data are: the area size of the target area (how clear is the algorithm about the target), the distance to the farthest position of the target region (how long may the trip take in worst case) or which significant borders may be crossed (e.g. tollgates).

IV. Discussion, Limits

Even though the approach is effective, fully implemented and provides surprisingly useful results, we have to discuss some limitations:

1. The current approach does not take into account the car's orientation. Usually, the orientation is obvious for a given route, but we can construct scenarios where we assume the wrong driving direction and finally a wrong target region. But if there was a means to measure the orientation, we could easily integrate it into the k and c^* functions.

2. Our target region is a set of crossings. For dense road networks this is a suitable model, but if we have large roads without any crossing, our target region border would only enclose the last crossing, even though a suitable target could be further away. To solve this problem, we had to evaluate t_{ex} values for outer crossings. This requires an adaption of our approach, as it currently explicitly avoids the processing of such crossings.

3. For the consecutive execution, we have to 'guess' a c_{max} value, as the future route (and its length) is unknown. If we go beyond the cost border, the target area will be empty. We can solve this with $c_{max}=\infty$, but this would require more memory, at least for the first g array.

4. We assume our positioning system can measure the position precisely enough to assign the correct road. If not, our approach may easily assume a target disoriented route which leads to wrong or empty target regions. As a solution, we may use a map-matching approach that even can be improved by our target orientation measurement. A basic precision of the positioning system (such as provided by GPS) is essential.

5. Our approach currently does not support multimodal routing. Consider what we call the 'park & ride problem': If the intention of a first trip is to start a second trip with another means of transportation, our approach may only predict the first target.

V. Conclusions and Future Work

We presented an efficient approach to predict a target region for a driven or walked route. The approach does not require a learning phase, thus can also be applied to routes that are driven for the first time. The output can geometrically be processed and used to query for significant objects in a spatial database. We can use this approach as a building block for different applications and services.

The next steps will address some current limitations: we not only want to collect crossings and links between crossings as the target area, but also part of roads. This however requires to restructure some algorithmic parts.

As a second goal: we also want to adapt the approach to improve the position measurement.

References

[1] M. Duckham, L. Kulik, M. Worboys and A. Galton, "Efficient generation of simple polygons for characterizing the shape of a set of points in the plane", Journal Pattern Recognition, Vol. 41, Issue 10, Oct. 2008, 3224-3236

[2] L. Chen, M. Lv, Q. Ye, G. Chen, J. Woodward, "A personal route prediction system based on trajectory data mining", Information Sciences 181 (2011) 1264–1284

[3] L. Chen, M. Lv, G. Chen, "A system for destination and future route prediction based on trajectory mining", Pervasive and Mobile Computing 6 (2010) 657–676

[4] V. Kostov, J. Ozawa, M. Yoshioka, T. Kudoh, "Travel Destination Prediction Using Frequent Crossing Pattern from Driving History", Proc. of the 8th Intern. IEEE Conf. on Intelligent Transportation Systems, Vienna, Austria, Sept. 13-16, 2005

[5] T. Terada, M. Miyamae, Y. Kishino, K. Tanaka, S. Nishio, T. Nakagawa, Y. Yamaguch, "Design of a Car Navigation System that Predicts User Destination", Proc. of the 7th Intern. Conf. on Mobile Data Management (MDM'06)

[6] R. Simmons, B. Browning, Y. Zhang, V. Sadekar, "Learning to Predict Driver Route and Destination Intent", Proc. of the IEEE ITSC 2006 Intelligent Transportation Systems, Toronto, Canada, Sept. 17-20, 2006

[7] K. Tanaka, Y. Kishino, T. Terada, S. Nishio, "A Destination Prediction Method Using Driving Contexts and Trajectory for Car Navigation Systems", SAC'09 March 8-12, 2009, Honolulu, Hawaii

[8] J. Krumm, "Real Time Destination Prediction Based On Efficient Routes", Microsoft Research, 2006-01-0811

[9] J. Froehlich, J. Krumm, "Route Prediction from Trip Observations", Microsoft Research, 2008-01-0201

[10] K. Miyashita, T. Terada, S. Nishio, "A Map Matching Algorithm for Car Navigation Systems that Predict User Destination", 22nd Intern. Conf on Advanced Information Networking and Applications – Workshops, AINAW 2008, 1551-1556

[11] J. Krumm, "A Markov Model for Driver Turn Prediction", Microsoft Research, 2008-01-0195

[12] F. Nakahara, T. Murakami, "A Destination Prediction Method Based on Behavioral Pattern Analysis of Nonperiodic Position Logs", 6th Inter. Conf. on Mobile Computing and Ubiquitous Networking, May 23-24, 2012 Okinawa, Japan

[13] S. Elnekave, M. Last, O. Maimon, "Predicting Future Locations Using Clusters' Centroids", ACMGIS'07, Nov. 7-9, 2007, Seattle

[14] J. Roth: "A Spatial Hashtable Optimized for Mobile Storage on Smart Phones", in M. Werner, M. Haustein (eds.): 9. GI/ITG KuVS Workshop Location based services and applications, Sept. 13-14 2012, Chemnitz, Germany, 71-84

[15] J. Roth: "Modularisierte Routenplanung mit der donavio-Umgebung", in M. Werner, M. Haustein (eds.): 9. GI/ITG KuVS Workshop Location based services and applications, Sept. 13-14 2012, Chemnitz, Germany, 119-131 (in German)

[16] J. Roth: "Combining Symbolic and Spatial Exploratory Search – the Homerun Explorer", Innovative Internet Computing Systems (I^2CS), Hagen, Germany, June 19-21, 2013, Fortschritt-Berichte VDI, Reihe 10, No. 826, 94-108

[17] J. Roth: "Die HomeRun-Plattform für ortsbezogene Dienste außerhalb des Massenmarktes", in A. Zipf, S. Lanig, M. Bauer (eds.) 6. GI/ITG KuVS Workshop Location based services and applications, Heidelberger Geographische Bausteine Heft 18, 2010 (in German)

[18] P. E. Hart, N. J. Nilsson, B. Raphael, "A Formal Basis for the Heuristic Determination of Minimum Cost Paths", IEEE Transactions on Systems Science and Cybernetics SSC4 (2), 1968, 100–107

[19] E. W. Dijkstra: "A note on two problems in connexion with graphs", Numerische Mathematik. 1, 1959, 269–271