International Journal of Knowledge-based and Intelligent Engineering Systems 20 (2016) 135–148
DOI 10.3233/KES-160339
IOS Press

Efficient many-to-many path planning and the Traveling Salesman Problem on road networks

Jörg Roth
Faculty of Computer Science, Nuremberg Institute of Technology, Nuremberg, Germany
E-mail: joerg.roth@th-nuernberg.de

Abstract. To compute optimal paths between locations in road networks is an important operation for e.g. transport business. Efficient algorithms for path planning usually assume a single start and a single target. In some scenarios, however, we want to compute *all* shortest paths from a set of starts to a set of targets. If we use the one-to-one planning in a loop, we consider each path as an independent computation. The drawback: we cannot take benefit of similarities between planning steps. In this paper, we present an approach that addresses this problem and provides a many-to-many path planning. An example that is heavily based on many-to-many path planning: given a home depot and a set of delivery points, what is the optimal round-trip that starts at the home depot, approaches each delivery point and turns back to the home depot. This problem is known as the *Traveling Salesman Problem*, but usually it is not applied to real road networks. In this paper we describe the efficient many-to-many path planning that is based on A*. It significantly reduces the number of visited crossings due to computation similarities. We adapt the approach to provide a solution for the Traveling Salesman Problem on road networks.

Keywords: Road transportation, shortest path problem, Traveling Salesman Problem

1. Introduction

Route planning on road networks is a well understood problem. Network services and on-board applications are able to compute routes from position to position. Resulting routes minimize 'costs' such as time, distance, fuel consumptions or road charges. Route planning algorithms usually are based on Dijkstra's shortest path approach or the A* algorithm. A* takes into account additional knowledge about road networks, modeled as future path cost estimation. With this, the computation is significantly faster while the optimality of results is kept.

A* is created to compute a route from a single starting point to a single target. For some scenarios, however, *all* routes from multiple starts to multiple targets are required. Examples:

— For a transport company we could request an optimal round-trip from the warehouse to all clients and back to the warehouse. This problem is known as the *Traveling Salesman Problem* (*TSP*). Even though we know algorithms that solve the TSP in practice, the algorithms assume that the distances between *all* pairs of target positions are known beforehand.

— For the more general problem, the so called *Vehicle Routing Problem* (*VRP*) [19], we have to deal with a fleet of vehicles, multiple depots and delivery points and want to distribute goods, while the overall transportation costs have to be minimized.

— Advanced *Geo Information Systems* (*GIS*) can be used to perform statistics on geo objects. More and more not only geometric conditions are considered (e.g. objects that are closer than 1 km to a location), but also the driving or walking distances. E.g. a GIS could be asked: '*present all pairs of hotels/conference sites that are in a walking distance of less than 10 minutes*'.

— If we use GPS to supervise driving, we have to map an inaccurate GPS measurement to a road position. Very often, multiple road positions could be possible for a single GPS measurement. A simple approach maps a position to the nearest road point. A more sophisticated approach would take multiple succes-

ISSN 1327-2314/16/$35.00

sive measurements, then compute the routes between all combinations of possible road points and select the route that fits best to the elapsed driving time.

All these examples need the many-to-many path planning as a basic operation. In principle, we could iterate through all pairs of starts and targets and execute the one-to-one-A*. Dependent on the respective implementation, execution platform, road network and routing tasks a single execution can take from 100 ms up to several seconds. Thus we have to face long executions for all combinations.

In this paper we introduce an efficient approach of multiple starts to multiple targets route planning. It is based on A* but the idea is to share planning results between iterations. The overall result is the same compared to the iterative execution of A*. Moreover, we take benefit of A* speed-up (compared to Dijkstra), based on the future path cost estimation and additional heuristics.

After describing the approach in detail, we present an important application of many-to-many routing: the Traveling Salesman Problem.

2. Related work

Virtually all route planning algorithms are based on Dijkstra's shortest path algorithm [6] or A* [9]. Both are similar – we can even consider the Dijkstra algorithm as a special case of A* if we set the future path cost estimation to zero. The idea of both algorithms: we model the road network as a graph of *nodes* (crossings between roads) and *edges* (road segments between crossings). Cost values assigned to edges reflect the property, an optimal route should minimize. It depends on the actual application to choose appropriate costs. Very often the time to drive through a road segment is used. This is based on the road type (e.g. urban, highway), road geometry (e.g. bent, straight, steep) and speed limit.

Both algorithms begin from the start node and consecutively visit further nodes that are connected to the currently visited area. '*To visit*' means: the total costs from start to this node are assigned and the algorithm knows the optimal route to this node. There is an important difference between the two algorithms: Dijkstra's algorithm visits *all* nodes with costs lower than the costs to the target. This means, also nodes are visited that reasonably cannot be part of the final route due to worst case considerations. In other words, Dijkstra tries to find the optimal route in *all* directions. A* visits far fewer nodes. Dependent on the future path cost estimation, A* visits only nodes that could in principle be part of the best route. Very often, the maximum speed on the beeline is used as estimation. Advanced approaches such as *Landmarks and Triangle Inequality* (*ALT*) [10] can provide a much better estimation, as they incorporate precomputed results into the estimation function.

For the one-to-one case, A* is more efficient than Dijkstra. On the other hand, as Dijkstra does not prefer a direction, it already provides the capability for one-to-many routing: we simply visit all nodes until all targets are included. For A* this approach would be crucial as the future path cost estimation is only suitable for a single target. This is one problem solved with our approach.

The idea to use Dijkstra for one-to-all planning is implemented in the *PHAST* approach [4]: it uses a Dijkstra approach without a stop condition, thus all nodes in the network are visited as potential target. To improve the execution speed, PHAST is based on so called *Contraction Hierarchies* (*CH*). A CH is a simplified road network that contains 'more important' nodes, e.g. nodes on fast roads. If two neighbors of a node take this node as shortest connections, the node can be removed and a new edge with the sum cost is added. The resulting simplified graph has to be computed beforehand. During route planning, the simplified graph is used as often as possible to reduce execution time.

The successor approach *RPHAST* [5] developed this idea for the one-to-many case: it reduces the CH graph to the respective targets. Similar approaches are *Hub Labeling* [1] or *Highway Hierarchies* [12] that are also based on a modified Dijkstra approach and incorporate similar techniques to simplify a road network graph.

The major drawback of Dijkstra-based approaches: even though the respective speed-up techniques ensure reasonable runtime behavior, the algorithms visit too many nodes that do not belong to result routes. This is because Dijkstra is not target-oriented in the way it selects the next node to visit. In contrast, A* uses the future path cost estimation that provides a target-oriented development of the visited area, while it remains optimal.

Another type of navigation approaches are based on *swarm intelligence*. As a basic idea: a number of individuals (often called *ants*), represent different alternatives to find a route to a target. Individuals are controlled by a cost function that attracts them to the target, often compared to *pheromone*. If the population is large, a huge number of alternatives is checked simultaneously, thus the result is close to optimality. Swarm intelligence is often applied in the area of robot routing [13] or communication networks, especially ad-hoc networks [18]. This is because the swarm intelligence is able to deal with uncertain environments that quickly change during execution. The drawback: the results depend on the random behavior of the individuals, thus is only an approximation. In our case, we have the perfect knowledge of the road network that does not change at runtime, thus we are able to achieve the theoretically optimal result. As a second drawback of the swarm approach: to deal with *multiple* starts and targets contradicts the idea of an ant colony where each ant deals with a *single* goal (e.g. to follow the virtual pheromone track). As a result we had to significantly increase to population to achieve appropriate results.

Many-to-many algorithms that base on the one-to-many path planning have an additional drawback: the respective iterations can be considered as isolated execution. But ideally, successive executions with nearby starting points should heavily benefit from former results with similar routes. Our novel approach addresses all these drawbacks and is fully built upon the A* approach, thus heavily benefiting from the target-oriented execution. Advanced optimization techniques for the one-to-one-A* (e.g. ALT estimation) can still be integrated to the many-to-many-case. In addition, techniques to simplify the road graph (e.g. CHs) can also be applied.

A problem that is usually ignored by related approaches is the problem of *crossing link displacement*: even though graph-based algorithms compute routes from crossing to crossing, in reality starts and targets are *not* on crossings, but are somewhere on a link between crossings. If we are able start a route in both driving directions (no one-way road), two entirely different routes may be the result. The same applies to the target. Many routing approaches either assume a known driving direction or simply take the geometrically nearest crossing. An ideal many-to-many routing approach should also solve the crossing link displacement problem.

3. One-to-one-A*

In order to present our multi-routing approach, we first describe the A* approach that computes routes from a *single* crossing to a *single* crossing. The later multi-routing approach heavily reuses A* whereas it avoids re-occurring computations. As mentioned above, we need a road graph: for each crossing q_i we know its *directly* connected crossings q_j and the driving costs $c(q_i, q_j)$ to get there. We call a connection between crossings q_i, q_j a *link*.

Costs $c(q_i, q_j)$ can be any non-negative numbers. In reality $c(q_i, q_j) \neq c(q_j, q_i)$ for many links due to, e.g., different speed limits or one-way roads.

The output route is a sequence of crossings ($start$, q_{i1}, q_{i2},… $target$) that minimizes the sum of link costs. In order to execute route planning in a target-oriented manner, A* additionally requires a future path cost estimation function $h(q_i, target)$ that provides a lower bound of costs for the route termination. The more h reaches the actual costs, the better A* performs.

Algorithm 1 presents the A* approach. Some remarks:

— $g[q_i]$ contains the currently *assumed* costs from *start* to crossing q_i.

— $f[q_i]$ contains the currently *estimated* total costs from *start* to *target*, if a route goes through q_i.

— We assign one of three states to each crossing: *not_visited*, *open* (g not finally computed), *closed* (optimal route from *start* discovered). In the algorithm above, '*expand crossing*' means: take an *open* crossing and check all its neighbors.

— To efficiently get the *open* crossing with the minimal f, we additionally need a priority queue *openList* that internally keeps the list sorted whenever an *open* crossing is added.

— If q_i is *closed*, $g[q_i]$ contains the minimal costs from *start* to q_i. If q_i is *open*, $g[q_i]$ may be still larger than the minimal costs.

— For *closed* crossings q_i, the link ($backLink[q_i]$, q_i) is the last link of the optimal route from *start* to q_i.

— Once we polled *target* from the *openList*, the optimal route is discovered. We then can easily collect the optimal route from *start*, following the *backLink* entries.

Even though the primary computation result is the *backLink* array, we also consider the *g* array as important output. For each crossing with state *closed*, the *g* array provides the minimal costs to get there from the start.

Note that *h* plays an important role for performance. Formally, the algorithm only requires *h* to be optimistic, but as mentioned earlier, good estimations can significantly reduce the number of visited crossings. In the following we consider *h* as a black box. We in particular do not require a special *h* function for our multi-routing approach. However, our approach will benefit from good *h* in the same way as A*.

ALGORITHM 1. One-to-one-A*

```
A_star(start, target)
  g[start]←0; f[start]←0; state[start]←open; openList.add(start);
  for all q_i≠start {
     g[q_i]← -1;  f[q_i]←0;  state[q_i]←not_visited;
  }

  do {
     q_i←openList.poll();  // get q_i with state open and minimal f[q_i]
     if q_i=target return success;
     state[q_i]←closed;

     for all neighbors q_j of q_i {          // 'expand' crossing
        if state[q_j] ≠closed {
           g_new←g[q_i]+c(q_i, q_j);
           f_new←g_new+h(q_j, target);
           if state[q_j]=not_visited or f_new<f[q_j] {
              g[q_j]←g_new;  f[q_j]←f_new;
              backLink[q_j]←q_j;
              state[q_j]←open;
              openList.add(q_j);
           }
        }
     }
  } while not openList.isEmpty();
  return failure;  // no route at all from start to target
```

4. Multi routing with EMMA

We now describe *EMMA* (*Efficient Many-to-Many-A**) to efficiently execute a route planning from multiple starts to multiple targets.

In principle we could execute the one-to-one-A* in two nested loops that iterate through all starts and targets. However, we can significantly speed up the execution, if we heavily reuse results from former iterations. We start with some observations:

— An ideal A* execution would only compute *closed* crossings that are on the optimal route. Even though we have *h* functions that are very close to the actual costs, typical A* executions expand 100 to 1000 times more crossings. This is information we could use both for other targets and other starts.

— A* iterates through *all* non-closed neighbors and checks, if the target can be reached more efficiently by this neighbor. If we knew the right neighbor from previous executions, we could significantly reduce the number of wrongly expanded crossings.

These observations lead to two basic mechanisms for many-to-many route planning namely *g array recycling* and *route-oriented expansion*. The EMMA execution is divided into two phases:

— The *Preparation Phase* takes a list of start and target links and computes the *g* and *backLink* arrays for all pairs of start/target.

— In the *Query Phase* we can query for optimal routes between any given start and target points. Each query is executed in a small *constant time*. We only require start and target points to reside anywhere on links passed to the Preparation Phase.

The idea is to run the more costly Preparation Phase *once* and compute routes in the Query Phase as often as required. Note that the preparation phase is executed for every new set of starts/targets. It should not be confused with a preparation of e.g. Contraction Hierarchies that is executed once for a road network.

In the following we describe the EMMA approach in detail. Even though we finally want to solve the many-to-many path planning problem, we in the following shift the view to one-to-many and many-to-one path planning to simplify the description.

4.1 The preparation phase

In the first step we take the list of start and target links that contain the later positions for route queries. Even though the start and target positions can be anywhere between crossings, the preparation is based on a crossing-to-crossing planning. The processing of starts and targets between crossings is shifted to the Query Phase.

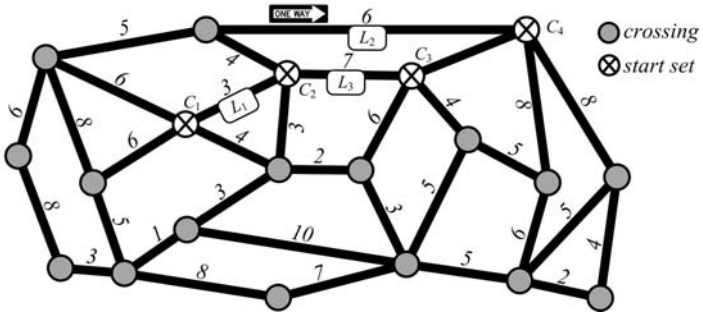

Fig. 1. Selection of crossings from start links

Figure 1 shows an example road network. Each link holds a cost value *seconds to drive through this link*; in further images we omit these values for better clarity. Apart from the one-way road L_2 we assume same costs for both directions.

We start with a set of three links $L_1 \ldots L_3$ that contain future starting positions of the Query Phase. Figure 1 illustrates how these start links are mapped to the respective crossings denoted by ⊗. Every link connects two crossings, but if a crossing occurs twice (e.g. C_2), it is only added once. If a link is one-way (e.g. L_2), only the next crossing in driving direction is considered (e.g. C_4). The mapping of target links works in the same way; we only use the opposite crossing for one-way links.

The caller of the navigation function can force a driving direction for any link, e.g. may request that a route starting at L_1 always must start in direction to C_2. This is useful, if the caller has knowledge about additional planning constraints. E.g., a car starting in a specific link may always park in a certain driving direction. We require this feature in our solution of the Traveling Salesman Problem later. The caller is free not to restrict the start and target link; then, both directions are considered.

4.1.1 One-to-many routing and *g* array recycling

We first assume, we only have *one* start and want to efficiently compute optimal routes to multiple targets. Having a deeper look at A* we can see, from the internal structures *state*, *f*, *g* and *backLink* only *f*

depends on the target. This is because f is the sum of g and h, and h is the cost estimation to the target. The other structures are assigned from the view of the start and never take into account the target. These are candidates to re-use for different targets.

Let $state_x$, f_x, g_x and $backLink_x$ denote the structures used for certain target $target_x$. For routes from *start* to two targets $target_a$, $target_b$ we get

— If $state_a$=$state_b$=*closed* then $g_a[q_i]=g_b[q_i]$ and $backLink_a[q_i]=backLink_b[q_i]$;

— But $f_a[q_i]\neq f_b[q_i]$ for most q_i.

If the route to $target_a$ was found, we are able to speed up the computation for $target_b$: we replace the initialization of the structures for $target_b$ as follows.

ALGORITHM 2. New Initialization

$g_b \leftarrow g_a$;
$state_b \leftarrow state_a$;
$backLink_b \leftarrow backLink_a$;
$openList_b \leftarrow openList_a$;
for all q_i in $openList_b$
  $f_b[q_i] \leftarrow g_b[q_i]+h(q_i, target_b)$
sort $openList_b$ according to f_b

Note that in reality, we do not copy large arrays in memory. E.g. as g_a is not used any longer, we simply can use it as g_b. We distinguish both targets in the pseudo code for better clarity.

Reusing the structures g_a, $state_a$ and $backLink_a$ for $target_b$ is obviously a reasonable approach as the second execution would compute the same values. Even though we reuse multiple structures, we call this approach the *g array recycling*.

This idea is illustrated in Fig. 2. The iteration to $target_1$ visits a certain set of crossings. The iteration b) can be executed in the former 'footprint', while only new crossings have to be visited. The best benefit shows c): the new target is a formerly *closed* crossing. This means the iteration can immediately finish, as the respective route is already known.

As the area of non-visited crossings gets smaller after each iteration, we have a good chance to save time for a new iteration. The benefit gets greater, if targets are either close together or targets are close to a formerly computed route. If a target is part of the computed g array for former targets, no computation is performed at all.

Reusing $openList_a$ is not obvious and needs explanation. As we first adapt the f values for all *open* crossings to the next target, these are correct. But we still have to argue, why the crossings in $openList_a$ also lead to a correct execution for $target_b$. Note that a plain execution for $target_b$ would start with an $openList$=(*start*).

We can prove the correctness of this approach as follows: The main *do* loop in the algorithm permanently modifies the $state_b$, g_b and $backLink_b$. We now have to find a possible sequence of steps in the original A* after which we also would get ($openList_b$, $state_b$)=($openList_a$, $state_a$). If there was such a sequence, the further execution (which then is unmodified A*) also must be correct.

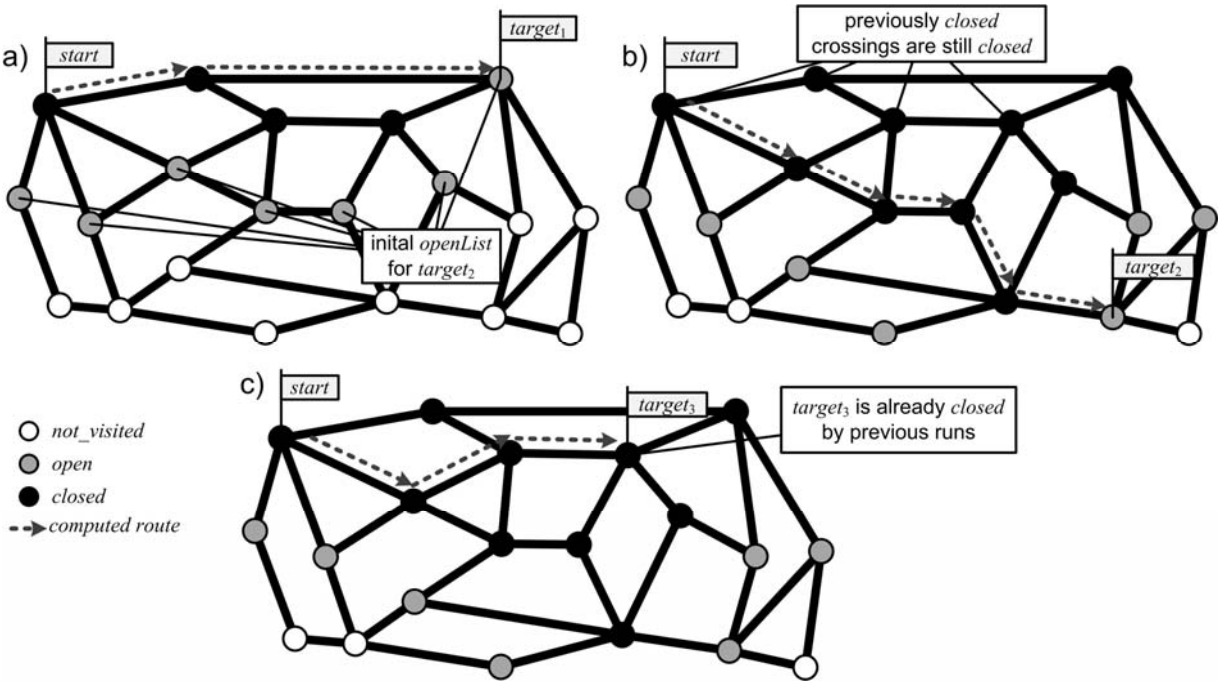

Fig. 2. Idea of the *g* array recycling

The only means to change the sequence of expanded crossings is the *h* function. We introduce a modified estimation function *h*' as follows:

$$h'(q_i, target) = \begin{cases} 0 & \text{if } state_a[q_i] = closed \\ h(q_i, target) & \text{otherwise} \end{cases}$$

Obviously *h*' still is optimistic, thus leads to correct results. Using *h*', the iterations first expand and close all crossing that also were *closed* for $target_a$. Due to the *h*' value of 0 no other crossings are *closed*, as the respective *f* value is minimal for all *open* crossings. As a result, the execution for $target_b$ with *h*' produces our initial values. In other words: our initial values could also be a result of an unmodified A* execution.

The only drawback of this procedure: *f* values of *open* crossings have to be recomputed for a new target. This means we have to sum the *g* entry (that is already computed) and the *h* function result (that from its nature has a quick execution) in a loop and reorder the *openList* due to the new *f* value.

One last point has to be clarified regarding the state of the target crossing: the original A* polls it from *openList* and terminates without modifying the state. This actually leaves an inconsistent entry: the crossing now is neither *open* nor *closed*. Now, as we recycle the *g* array, we have to be more careful. We can simply solve this, if we just re-insert the target to *openList* on termination. As the target's neighbors have not been visited, the target crossing must remain *open*.

Figure 3 shows an execution of the one-to-many case on real data. The road network contains 11 million crossings and 28 million unidirectional links. We subsequently performed a route planning to four targets. The dark regions show the area of newly visited crossings after an iteration. The light areas indicate the crossings that would have *additionally* been visited if we used the one-to-one-A*. We can easily see: compared to the one-to-one-A* our approach usually visits a smaller amount of new crossings. For, e.g., $target_2$ and $target_4$ nearly all crossings already have been expanded during a former iteration and only a very small amount has to be newly visited.

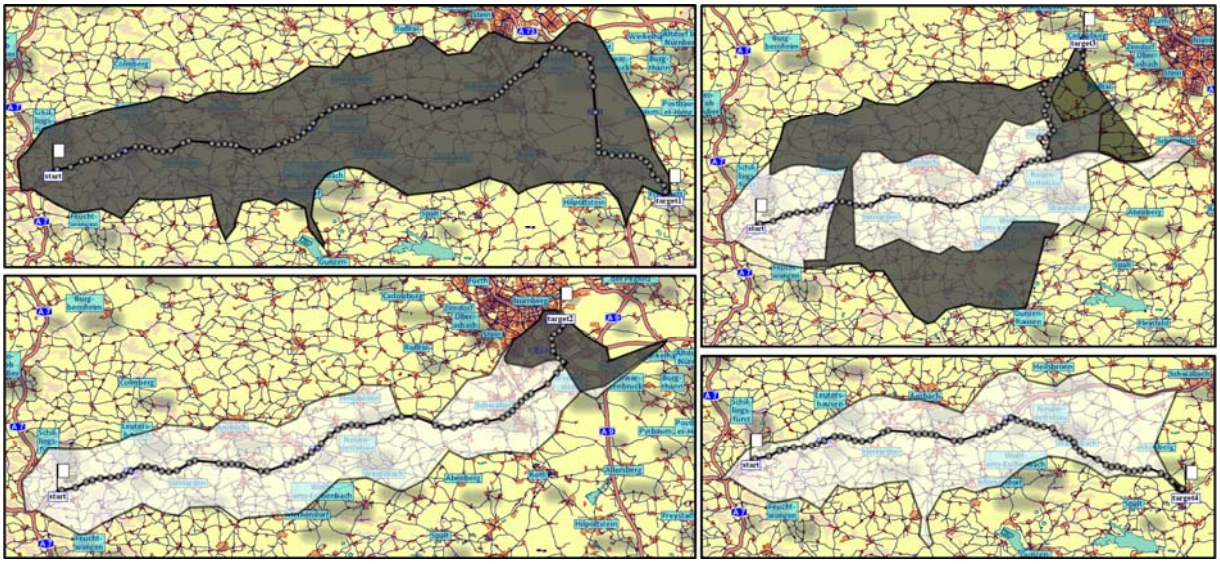

Fig. 3. The one-to-many case in a real road network (newly visited crossings, dark regions and crossings that one-to-one had visited, light regions)

Note that the regions are only a graphical representation of a set of crossings. We used a concave hull function [8] to present the set of crossings by closed polygons for better clarity.

If we planned all four routes separately in this example, we would have visited 52203 crossings. With g array recycling we only visited 24805 crossings. This is a speed-up factor of 2.1. We have to recompute the f values of 5629 crossings.

The improvement would get larger for more targets, especially, if some of them are close together. The theoretic boundary value is a speed-up by factor n for n targets.

4.1.2 Many-to-one routing and route-oriented expansion

The second optimization takes the view of a single target. The idea is to store optimal routes from former starts to this target. Whenever we expand any crossing that already resides on a formerly stored route later, we already know the optimal route. We can use this to significantly save computation time.

Figure 4 illustrates the idea. In a) the route from $start_1$ to *target* is discovered. In b) the search goes from $start_2$ until the crossing marked with '*' is visited: here, we do not have to add all neighbors to *openList*, but only the next hop to target marked with '!' as this *must* be the only appropriate neighbor. The same applies to $start_3$ in c). We call this *route-oriented expansion*, which can be summarized as follows:

— Whenever we found a route, we store the next hop neighbor for all crossings that belong to this route (presented by solid arrows in Fig. 4). We accumulate these entries for further starts.

— Whenever we poll a crossing from *openList* that holds such a stored direction, we modify the expansion: We do not iterate through all neighbors but only take the next hop neighbor that is formerly stored. For this neighbor we proceed as usual, i.e. we open it (if not already) and modify the g and f entries.

It is important to note that this improvement does not affect correctness and optimality. We are not forced to use the existing route from all starts; alternative (potentially better) routes still can 'win' against existing routes. But it requires fewer visits to discover such solutions because for an increasing number of crossings, only a single neighbor has to be expanded. The benefit gets greater for more starts as the probability to expand into an existing route gets higher.

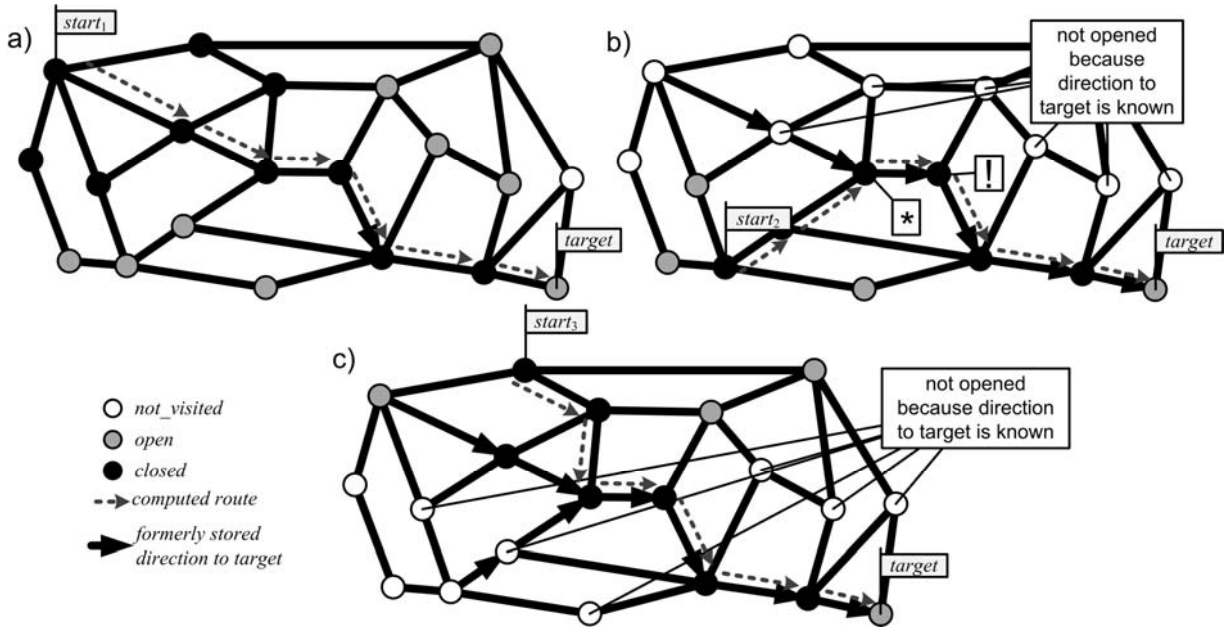

Fig. 4. Idea of route-oriented expansion

We can show the correctness for the approach as follows: Let R_t be the sequence of crossings of the formerly computed routes from $start_t$. To compute a new route R_s, our route-oriented expansion ignores links that leave any former route. We can express this by a modified cost function c'_s:

$$c'_s(q_i, q_j) = \begin{cases} \infty & \text{if } \exists t < s : q_i \in R_t \text{ and } q_j \notin R_t \\ c(q_i, q_j) & \text{otherwise} \end{cases}$$

We have to ensure that the optimal route from $start_s$ does not contain a link (q_i, q_j) with $c'_s(q_i, q_j) = \infty$. This obviously cannot happen, as q_i was part of an optimal route to the same target (though another $start_t$) and the optimal route did not go through q_j. In other words: two routes from different starts and same target cannot cross – they have to be identical from the first shared crossing (that may at least be the target).

Figure 5 shows the route planning for four starts to a single target. The dark areas show the visited crossings for route-oriented expansion. The light areas again illustrate the additional crossings that would have been visited for the one-to-one-A* approach. We can easily see the benefit for $start_2$…$start_4$. As the western part of the routes is identical for all starts, route-oriented expansion omits the expansion of the respective neighbor crossings. In contrast to the *g* array recycling, this approach does not require any recomputation.

Some statistics: If we planned all four routes separately, we had to visit 17105 crossings. Using route-oriented expansion we only visited 9650. This is a speed-up factor of 1.77.

Similar to the *g* array recycling, the speed-up depends on the respective routing task. The improvement gets larger for more starts, especially, if some of them are close together. The boundary value is a speed-up by factor *n* for *n* starts.

4.1.3 Many-to-many routing and the shielded target problem

The *g* array recycling and route-oriented expansion both independently work correctly, i.e., the computed routes are theoretical optimal. The next step is to put both together:

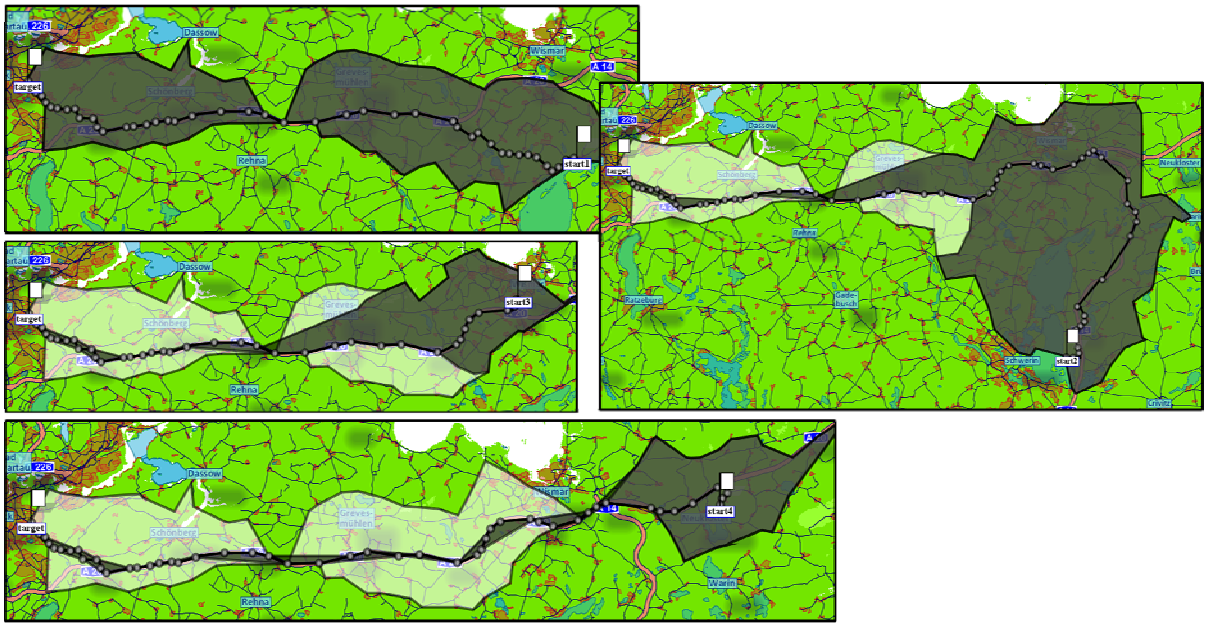

Fig. 5. The many-to-one case in a real road network

— The main loop iterates through the list of starts.

— For each start, we route to all targets using the *g* array recycling.

— We store all routes to a target and apply route-oriented expansion whenever possible.

On the first glance, this approach is reasonable and takes the benefit of both methods. However, we get a new problem that only occurs in combination – we call it the *shielded target problem*.

Figure 6 presents a scenario with 3 starts and 2 targets. We assume the iterations for $start_1$ and $start_2$ to the two targets already have been finished and we now execute the iteration from $start_3$ to $target_1$. At the time we visited the crossing marked with '*' we use route-oriented expansion with the former route from $start_1$ to $target_1$. This iteration terminates correctly.

If we now want to plan from $start_3$ to $target_2$, the *g* array recycling leads to a problem: the crossing marked with '!' is still *not_visited*. This was, because the crossing '*' prevents its opening due to route-oriented expansion. As a result, the link from '*' to '!' cannot be part of a computed route any more for further targets. In this case, either only suboptimal routes are found, or in some cases, no route at all will be discovered.

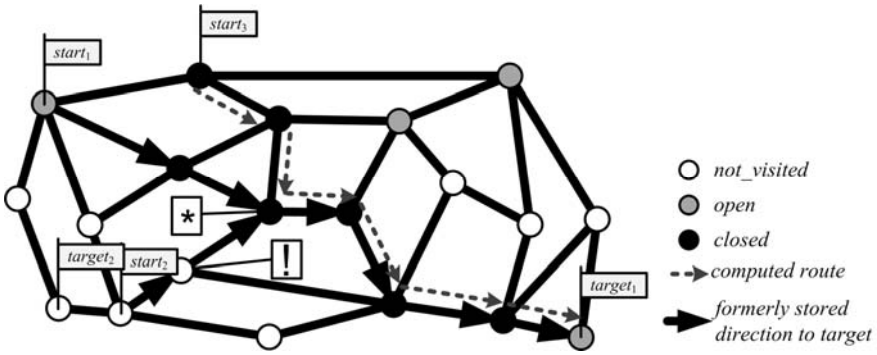

Fig. 6. The shielded target problem

As a solution, we introduce a second *open* state.

— During an expansion step: whenever neighbors are filtered out due to route-oriented expansion they are marked as *open_not_followed*. They not will appear in the *openList*.

— If for the next target we apply *g* array recycling, the new *openList* is the union of the former *openList* and the former list of *open_not_followed* crossings.

— To easily collect all crossings that are *open_not_followed*, we introduce a new list *openListNotFollowed*. As it is not required to quickly retrieve special objects (such as for the *poll* operation), we can use a simple container structure (e.g. hash set) for this list.

Figure 7 shows the corrected execution.

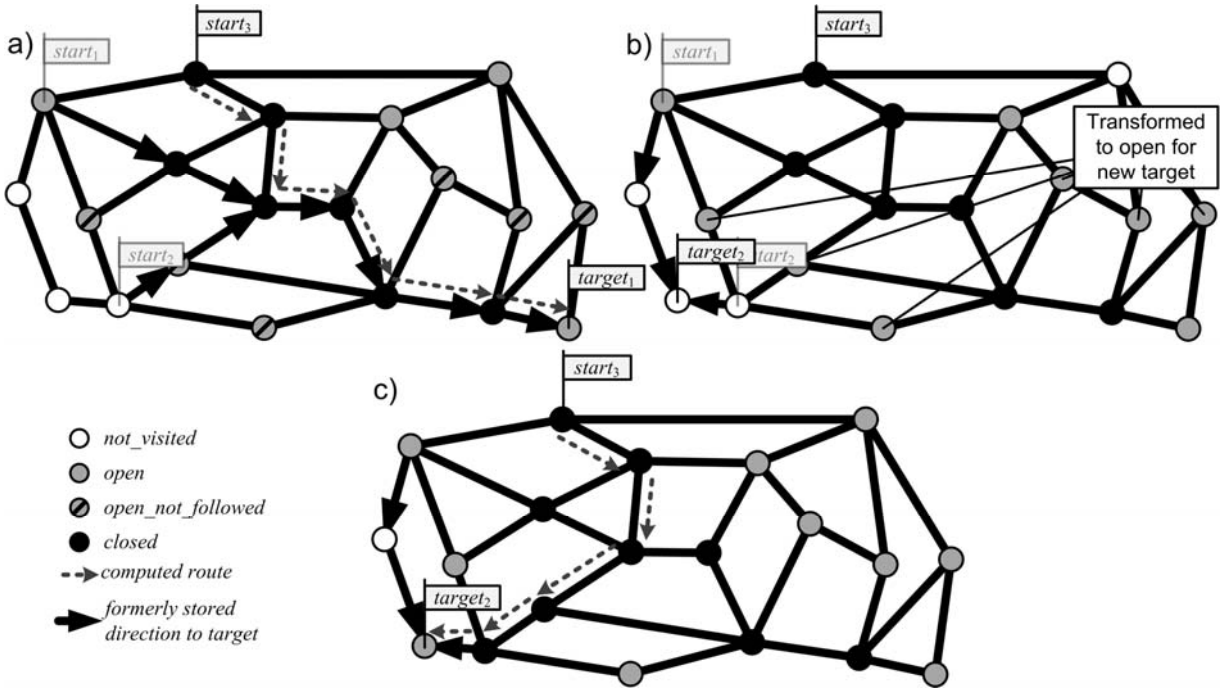

Fig. 7. Solution of the shielded target problem with a second *open* state

In a) we see the normal execution, but some crossings are stated as *open_not_followed*. These are in b) joined with the former *openList* to start a route to $target_2$; c) shows the final, correct result.

4.2 The query phase and the link displacement problem

The situation after the Preparation Phase terminated: from all involved start crossings we know the optimal path and costs to all involved target crossings. The Query Phase allows querying for start and target positions between crossings, i.e. the Query Phase mainly solves the *crossing link displacement problem*.

As mentioned before, this problem is usually ignored, even though it must be addressed by real applications. Especially if we have long road segments, a simple approximation (e.g. map the position to the nearest crossing) may result in suboptimal routes.

An ad-hoc solution would temporarily add the requested locations as new start and target crossings to the road network and splits up the respective links. Theoretically easy, this solution may not be applicable in real route planning applications. The reason: usually the road network is stored in a highly optimized data structure that both stores topological as well as geometric information. The latter is used for map display or to generate navigation commands. In addition the structure contains spatial index information and undergoes compression procedures to safe memory. Thus, it usually is critical to integrate new start and target positions.

EMMA solves this problem without the need to change the road network. Moreover, the Query Phase allows to query for routes from *any* starting and to *any* target point on previously registered links in *constant time*.

Figure 8 shows a link between the crossings *A* and *B*. The real position *Pos* resides anywhere between these crossings. We assume the position is already mapped to the road's linestring geometry. We further assume, there exists a function ℓ that can compute the real length of line string segments. This function has to compute the length in an appropriate scale unit. E.g. if costs are base on speed, we have to process a length in meters or miles and but *not* degrees. As usually linestring geometries on the Earth's surface are stored in latitude/longitude degrees, a geodesic mapping is required.

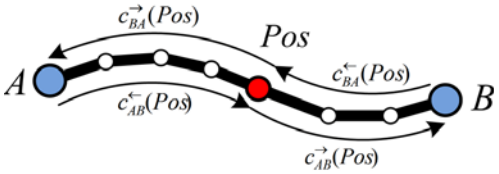

Fig. 8. Costs of a segmented road

For a link *AB* with two driving directions, we got a total of four directed segments with respective partial costs: two outgoing costs $c^{\rightarrow}_{AB}$, $c^{\rightarrow}_{BA}$ (used for starts) and two ingoing costs $c^{\leftarrow}_{AB}$, $c^{\leftarrow}_{BA}$ (used for targets). Partial costs may base on motion models that e.g. consider acceleration. In most cases a linear relation is sufficient, i.e.

$$c^{\leftarrow}_{xy}(Pos) = \frac{\ell(x, Pos)}{\ell(x, y)} \cdot c(x, y)\,,\; c^{\rightarrow}_{xy}(Pos) = \frac{\ell(Pos, y)}{\ell(x, y)} \cdot c(x, y)\,.$$

If we have a start link *AB* and a target link *CD*, we need four comparisons as shown in Figure 9 to find the best route.

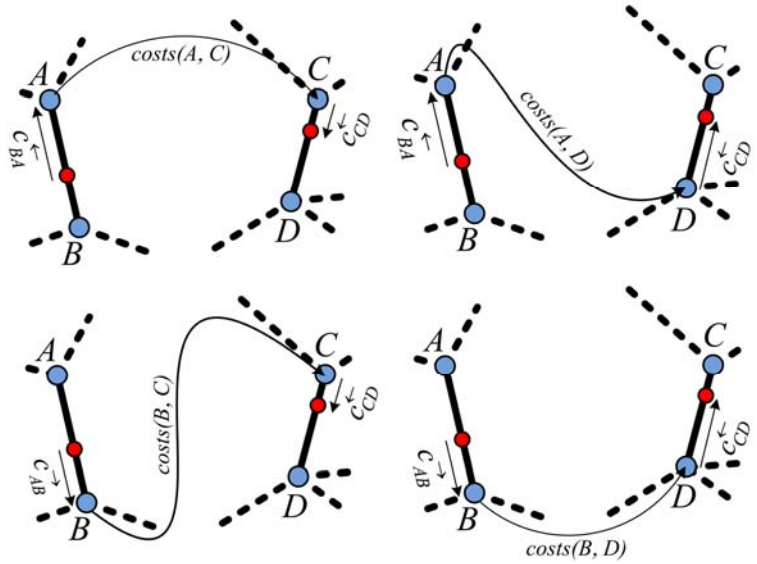

Fig. 9. Comparisons to find the optimal route between link positions

Some remarks:

— The costs for *AB*, *BA*, *CD* and *DC* are stored in the road network, thus the four partial costs ($c^{\rightarrow}_{AB}$, $c^{\rightarrow}_{BA}$, $c^{\leftarrow}_{CD}$, $c^{\leftarrow}_{DC}$) can instantly be computed. The route costs from crossing to crossing (i.e., *costs*(*A, C*), *costs*(*A, D*), *costs*(*B, C*), *costs*(*B, D*)) are already computed in the Preparation Phase. Thus the four comparisons can be executed with only few instructions.

— We may reduce the four comparisons to two or even one, if links have a restricted direction. This can either be because they are one-way roads or the caller of the query restricted the starting or target direction due to additional knowledge. E.g. the caller may know the parking position, thus only allows starting in a certain direction.

— We have to deal with the special case *AB*=*CD* or *AB*=*DC*. This means we have a road segment between two positions on the same link. This does not necessarily mean the shortest path is *on* the respective link, but this path has explicitly to be taken into account as a potential solution.

The caller of the EMMA multi-routing now easily is able to query all pairs of start and target positions to get the respective optimal route.

5. Traveling salesman on real road networks

We now apply the solution to the well-known *Traveling Salesman Problem* (*TSP*) [11]. One important use case is logistics and carrying business. The problem is to find an optimal round-trip to a given set of locations. E.g. consider the locations are the home depot and a set of delivery points: the problem is to find the optimal (e.g. shortest) round-trip that starts and ends at the depot and visits each delivery point once. Similar examples are

— Mail: Find a round-trip for a postman to all houses;

— Sightseeing tours: A tourist bus has to visit all sights of a city and drive back to the hotel.

Solutions have to execute two tasks:

— Discover a matrix of costs between all pairs of targets;

— Find a sequence of locations that minimizes the sum of costs.

The second task is subject of intensive research. The problem to find the theoretical optimal sequence is *NP-hard*; no polynomial-time solution is known. Fortunately, there exist good approximations that either produce optimal or acceptable suboptimal round-trips.

5.1 Computing the cost matrix

The first problem usually is ignored by researchers in the TSP area – the cost matrix usually is considered as given. But if we take a real road network, the cost matrix is based on optimal route finding between *all* pairs of targets. All routes have to be computed beforehand. As for n targets, we have $n^2 - n$ routes; this can be a time-consuming task.

In principle, we also could estimate the costs between targets. Such an approach is presented in [7]: we could e.g., use the line of sight distance and multiply it with a locality-dependent factor. But we easily can find counter-samples where such estimations completely fail – imaging geographically nearby targets on opposite river banks without a bridge. We strongly believe, the final result heavily depends on the exact cost matrix. This is the motivation to take the EMMA approach as it efficiently can compute the costs between all round-trip locations.

Research on the TSP distinguishes the *symmetric* and *asymmetric* TSP. Most assume symmetric costs, i.e. *costs*(*A, B*)=*costs*(*B, A*) for all locations. This is obviously not true in real road networks. [17] pointed out that if we apply symmetric TSP approximations to the asymmetric road network scenario, this may lead to significantly different solutions.

Looking deeper to the problem, real road networks lead to another problem usually ignored by TSP researchers: we have to consider the direction of arrival and departure. For non-one-way roads we usually

can approach a certain target from two directions. Dependent on the respective situation we can distinguish three cases (Fig. 10):

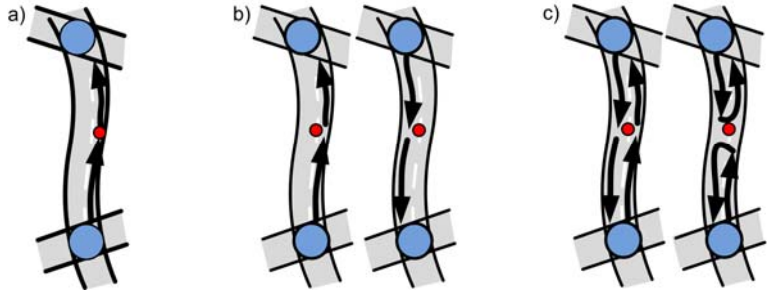

Fig. 10. Cases for arrival and departure directions

— It could be required to arrive in a certain direction *and* depart in the same direction (Fig. 10(a)). This could e.g. be if the actual point to visit can only be reached from one side of the road.

— The target could be reached from both sides, but we could request to depart in the same direction (Fig. 10(b)).

— Some targets could also accept both arrival directions and U-turns to continue the round-trip (Fig. 10(c)).

Our additional goal was to assign one of these cases to each target. Note that some scenarios cannot avoid U-turns: consider a target in a dead-end. However: U-turns cause certain costs that have to be reflected in the cost matrix.

Based on these considerations, the typical n x n cost matrix is not sufficient any more as we have to compute up to four costs per pair of targets. In worst case, we thus get a total of $4 \cdot (n^2 - n)$ routes. For, e.g., 50 targets, we have to compute 9800 routes.

5.2 Approximating the round-trip

The main contribution of this paper is the many-to-many routing approach and not the TSP approximation. But we had to consider some details in order to bring both together. As we have asymmetric costs and also respect arrival and departure directions, most TSP approximations were not suitable: most solutions assume symmetric costs and completely ignore directions. We eventually decided to choose a *Simulated Annealing* (*SA*) approach [3] as it was able to address all our requirements. The basic idea of SA:

— We first have to define a space of *states*; for our problem the states are all possible round-trips. A *fitness function* indicates how 'good' a state is – for round-trips it is based on the total costs.

— Each state is expected to have *neighbor* states (see below). Starting from a random state we take a sequence of random neighbors that increases the fitness value (*hill climbing*).

— With a certain probability (that decreases over time), we choose a neighbor with lower fitness value. This ensures not to get trapped in a sub-optimum.

To complete our description, we have to define what a *neighbor state* is. Note that SA leaves it to the actual realization to choose an appropriate notion of neighbors. Our result was outcome of numerous experiments. For a round-trip, neighbors are:

— If we reverse a subsequence of targets. E.g. for the sequence *ABCDEFGHA*: neighbor states are *ABC**FED**GHA* or *A**GFEDCB**HA*;

— If we keep the sequence of targets, but change one arrival or departure direction.

With a probability of 75% we choose a random neighbor of the first type and with 25% of the second type.

5.3 Results and evaluation

For tests and evaluation we implemented the approach in our *donavio* navigation environment [16] that is part of the *HomeRun* framework [15]. donavio is a technically mature platform that contains all types of components related to navigation services:

— Import, editing, optimizing, compiling and compressing of road network data; we especially are able to compute road networks solely from a geometric representation, such as of *OpenStreetMap* [2];

— Functions to model different means of transportation;

— Data structures with indexing technologies for speed and space optimization;

— Navigation algorithms with speed-up techniques (such as ALT);

— Support for offline navigation on mobile platforms [14];

— Generation of driving commands (such as '*at next crossing turn left*');

— Route display on maps.

As we developed the donavio environment ourselves, we are able to integrate and test arbitrary extensions and modifications at any part of the processing chain. We fully implemented the EMMA approach as well as the TSP approximation in our donavio navigation environment. From our geodata we take the road network of Germany with 12.3 million links, 10.8 million crossings and a total of 737 thousand kilometers roads. From the plenty of tests we conducted, we present 6 representative scenarios:

— *TSP1*: 9 targets in an area of 880 km x 550 km, 6 targets in different cities, 3 targets in one city;

— *TSP2*: 34 targets in a single city in an area of 16 km x 11 km; a 'packet delivery' scenario;

— *TSP3*: 15 cities in an area of 280 km x 230 km, each city contains 3 targets;

— *TSP4*: another 'packet delivery' scenario; a single target in one city (the 'main depot'), 29 targets in a second city (the 'delivery points'), 160 km distance between the cities;

— *TSP5*: 5 cities in an area of 160 km x 100 km, each city contains 10 or 11 targets with a total of 51 targets;

— *TSP6*: 2 cities with a distance of 650 km, each city contains 20 targets.

These scenarios reflect realistic tasks for, e.g. packet delivery or carrying business. Please note that random tests are not reasonable to evaluate EMMA, as this approach benefits from route similarities only found in real world scenarios.

Figure 11 shows three of the results on a map. Table I provides an overview of the runtime statistics. Column I shows the number of multi-routing targets. A first observation: the results cover a huge variety of different numbers. This is a typical observation, whenever navigation algorithms are analyzed: results heavily depend on the respective scenarios. This makes it difficult to numeralize a benefit, as it can get larger or smaller, dependent on the actual situation. E.g. the effects of our approach in *TSP1* and *TSP2* are entirely different: in larger areas with only view targets (*TSP1*) the benefit is smaller, whereas in smaller areas with many targets (*TSP2*) we have a 100 times shorter execution time.

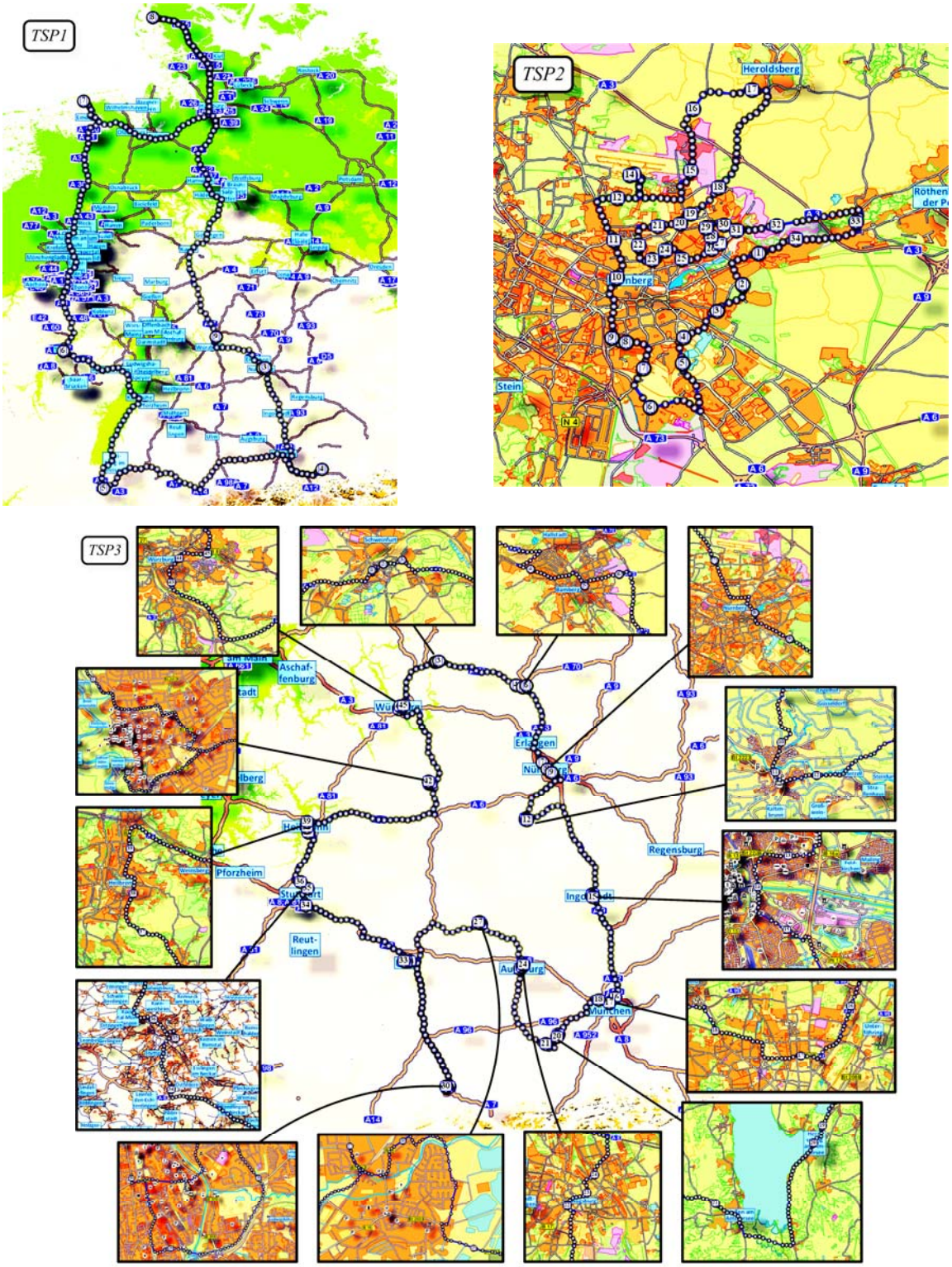

Fig. 11. Three round-trips

Table I Runtime Statistics to compute routes for the Traveling Salesman Problem

Test	I Number of TSP targets	II Visited crossings *EMMA* (in mil.)	III Visited crossings *one-to-one-A** (in mil.)	IV Relation of III/II	V Recomp. *f* values (in mil.)	VI Recomp. time of *f* values/total time (in %)	VII Total Time *EMMA* (in ms)	VIII Total Time *one-to-one A** (in ms)	IX Relation of VIII/VII
TSP1	9	8.3	15.5	1.9	1.6	16.1	112.3	224.9	2.0
TSP2	34	0.3	1.3	5.2	0.1	14.2	0.3	30.6	103.4
TSP3	45	11.4	79.5	6.9	7.5	32.1	9.3	62.6	6.7
TSP4	30	1.3	9.4	7.1	0.4	15.5	1.8	39.2	21.8
TSP5	51	5.3	54.7	10.4	3.7	29.2	3.0	45.1	15.2
TSP6	40	11.5	151.4	13.2	11.5	46.2	14.8	107.1	7.2

Due to the different results, formal analyzes to access the runtime complexity *O(…)* are difficult. As an important measure, we counted the number of visited crossings by EMMA (column II) compared to the one-to-one-A* in a loop (column III). The reduction factor (column IV) indicates how many fewer crossings are visited due to *g* array recycling and route-oriented expansion. The results are encouraging: in our examples, in worst case we have only visited about half of the amount of crossings, in best case only 1/13.2. This is a significant benefit.

The drawback of the *g* array recycling was to recompute *f* values of all *open* crossings (see Section 4.11, especially Algorithm 2). We measured the amount of affected crossings (column V) and the relative processing time (column VI). Certain crossings can undergo an *f* value recomputation multiple times. In our table, we thus counted not only the crossings, but the actual number of recomputations. Not surprisingly, the number of recomputed crossings thus is high; in *TSP6* the number is as large as the total visited crossings. One could argue that the amount of time spent for *f* values recomputation is high. But this is a result of the dramatic reduction of the overall visited nodes.

Column VII presents the mean time for each computed route for the Traveling Salesman Problem, measured on an Intel i7-4790 3.6 GHz. The execution times also heavily vary. As they depend on processor, memory and implementation details they only should give an impression. Column VIII shows the time for the same tests, if we used the naïve one-to-one-A* in a loop. Column IX shows the speed-up factor of our approach compared to the naïve approach. The comparison does not only take into account the relation of visited nodes, but also the constant set-up time for each route, e.g. to deal with the crossing link displacement. But also the time to sort the priority queue *openList* significantly differs between the tests. Thus, the speed-up indicated by column IX is usually different from the relation indicated by column IV.

To sum up: even though, we have additional costs due to *f* value recomputation, the approach pays off and significantly speeds up the computation or routes required to solve the Traveling Salesman Problem by essential factors.

6. Conclusion

In this paper we presented an optimization to A* to solve the many-to-many path planning problem called *EMMA*. In contrast to a one-to-one execution in a loop, we heavily benefit from similarities in the respective computation steps. For this, we introduce two techniques: *g array recycling* and *route-oriented expansion*. With these, we heavily reduce the number of visited crossings. Important: even though EMMA speeds up the computation time, it still provides correct, optimal routes as computed by one-to-one planning. Our approach can incorporate additional A* speed-up techniques such as ALT estimations to further improve computation time.

We used our many-to-many path planning to solve the Traveling Salesman Problem in road networks. While most solutions assume a given cost matrix, we are now able to effectively compute it. Moreover, our approach solves the *crossing link displacement problem* and considers departure and arrival directions. We used a Simulated Annealing algorithm to actually compute round-trips from the cost matrix.

References

[1] I. Abraham, D. Delling, A. V. Goldberg and R.F. Werneck, A hub-based labeling algorithm for shortest paths on road networks, in: *SEA*, LNCS 6630, Springer, (2011), 230–241.
[2] J. Bennet, Open street map, Packt Publishing, 2010
[3] D. Bertsimas and J. Tsitsiklis, Simulated annealing, *Statistical Science* **8** (1) (1993), 10–15.
[4] D. Delling, A. V. Goldberg, A. Nowatzyk and R.F. Werneck, PHAST: Hardware-accelerated shortest path trees, in: IPDPS, IEEE Computer Society (2011).
[5] D. Delling, A. V. Goldberg and R.F. Werneck, Faster batched shortest paths in road networks, in: *Proc of the 11th Workshop on Algorithmic Approaches for Transportation Modeling, Optimization, and Systems* (*ATMOS'11*), Open Access Series in Informatics, (2011), 52–53.
[6] E.W. Dijkstra, A note on two problems in connexion with graphs, *Numerische Mathematik* **1** (1959), 269–271.
[7] N. Dubois and F. Semet, Estimation and determination of shortest path length in a road network with obstacles, *European Journal of Operational Research* **83** (1995), 105–116.
[8] 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*, **41**(10) (Oct 2008), 3224–3236
[9] P.E. Hart, N.J. Nilsson and B. Raphael, A formal basis for the heuristic determination of minimum cost paths, *IEEE Transactions on Systems Science and Cybernetics SSC* 4(2) (1968), 100–107.
[10] A.V. Goldberg and C. Harrelson, Computing the shortest path: A* search meets graph theory, in: *Proc 16th ACM-SIAM Symposium on Discrete Algorithms* (2005), 156–165.
[11] G. Gutin and A.P. Punnen, eds., The traveling salesman problem and its variations, Kluwer, 2002.
[12] S. Knopp, P. Sanders, D. Schultes, F. Schulz and D. Wagner, Computing many-to-many shortest paths using highway hierarchies, in: *ALENEX, SIAM*, (2007), 36–45.
[13] D.R. Parhi, J.K. Pothal and M.K. Singh, Navigation of multiple mobile robots using swarm intelligence, *World Congress on Nature & Biologically Inspired Compuing NaBIC*, (2009), 1145–1149.
[14] J. Roth, A spatial hashtable optimized for mobile storage on smart phones, in: *9. GI/ITG KuVS Workshop on Location-based Services*, M. Werner and M. Haustein, eds., 13–14 Sept 2012, TU Chemnitz, 2013, 71–84.
[15] J. Roth, Combining symbolic and spatial exploratory search – the homerun explorer, *Innovative Internet Computing Systems* (*I2CS*), Hagen, VDI, **10**(826) (19–21 June 2013), 94–108.
[16] J. Roth, Predicting route targets based on optimality considerations, *Intern Conf on Innovations for Community Services* (*I4CS*), Reims (France), IEEE xplore, (4–6 June 2014), 61–68.
[17] A. Rodrígueza and R. Ruiz, The effect of the asymmetry of road transportation networks on the traveling salesman problem, *Computers & Operations Research* **39**(7) (July 2012),1566–1576.
[18] C.-C. Shen and C. Jaikaeo, Ad hoc multicast routing algorithm with swarm intelligence. ACM Mobile Networks and Applications Journal **10** (2005), 47–59.
[19] P. Toth and D. Vigo, eds., The vehicle routing problem. *SIAM – Society for Industrial and Applied Mathematics*, (2002).