Efficient Computation of Bypass Areas

Jörg Roth

Abstract Route planning in road networks is a basic operation in the area of location-based services. Very often, the knowledge of the optimal route is not the only important information for a driver. Complex services could also present points of interest (e.g. hotels or gas stations) nearby the optimal route as stop-over. Here, 'nearby' means: the bypass route from a start to target that passes that point does not exceed certain costs. In this paper, we present an efficient approach to compute all bypasses that are within a given cost limit. We may additionally request only locally optimal bypasses, e.g., that reach an intermediate point without driving U-turns. The set of all bypasses called *bypass area* can be used for further queries, in e.g. geo databases to find nearby points of interest for a certain application or service. Our approach is fully implemented and evaluated and computes the respective bypass areas very runtime-efficient, whereas it re-uses similar structures as for optimal route planning.

Keywords Route planning · A* · Alternative routes · Bypasses

1 Introduction

Route planning on road networks is a well understood problem. Network services or on-board applications are able to compute a route from position to position that is optimal according to given demands. Resulting routes minimize 'costs' such as time, distance, fuel consumptions or road charges.

For a certain navigation task, the optimal route may not be the only useful information. A driver could refuse the optimal route due to several reasons. From former knowledge, the driver may know the given route does not meet her or his

J. Roth (✉)
Department of Computer Science,
Nuremberg Institute of Technology, Nuremberg, Germany
e-mail: Joerg.Roth@th-nuernberg.de

G. Gartner and H. Huang (eds.), *Progress in Location-Based Services 2016*, Lecture Notes in Geoinformation and Cartography,
DOI 10.1007/978-3-319-47289-8_10

demands on an optimal route, e.g., there is a certain probability for a traffic jam. Or, the driver wants to drive through another region to view a certain sight.

On the other hand, the route planning process should directly incorporate a certain intermediate point. e.g., we want to drive to a target, but on the route we want to stop at an arbitrary gas station. Driving to downtown we may ask: how to drive, if we want to pass a post box, bakery and supermarket? Important: the actual intermediate positions are variable (*any* post box or *any* bakery will do and also any ordering of stops). Thus, we cannot simply split a route and execute individual point-to-point planning tasks to known points. Thus, we have to compute *all* routes from a start to a target and check, whether they touch the respective points of interest.

The following approach to solve this problem may be part of a larger navigation service or route planning application. We present an approach that efficiently computes all respective bypasses. The approach can be configured to reflect the user's or application's demands. In particular we express the quality of a bypass with the help of a cost limit.

2 Related Work

Route planning algorithms usually are based on Dijkstra's shortest path approach (Dijkstra 1959) or the A* algorithm (Hart et al. 1968). 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. For large road networks, we usually combine the benefits of A* with additional techniques to reduce the runtime. They can be distinguished in approaches that do not change the optimal result (i.e. only speed up the computation) and those that may lead to sub-optimal results. An approach of the first type is to provide precise future path cost estimations, e.g., *ALT* (*A* search, landmarks, and triangle inequality,* Goldberg and Harrelson 2005). An approach of the second type is to consider only fast roads (e.g. highways), in the middle of the route, i.e. when we exceed a certain distance from start or target (Geisberger et al. 2008).

Bypasses or alternative routes are *non*-optimal routes. Usually, there is a *single* optimal route for a certain cost measure (if we ignore the unlike case of multiple identical minimal costs). In contrast, the number of routes with non-optimal costs is unlimited. In order to limit the set of reasonable bypasses, we thus have to introduce additional conditions on bypasses.

An approach is to order all possible routes by their increasing costs and take the first k routes. The so-called *k-shortest path routing* has a long tradition (Hoffman and Pavley 1959; Bellman and Kalaba 1960); several subsequent approaches speed up the execution (Yen 1971; Eppstein 1994). The problem turned out to be harder, if we want to avoid routes with loops—but this is usually a condition for a reasonable bypass.

Abraham et al. (2010) pointed out a problem of the k-shortest path approach: reasonable alternatives are probably not among the first thousands shortest paths. They suggest an additional condition for alternative routes: *local optimality*. This means, every subroute up to a certain length must be an optimal route. This reflects usual driving behavior: even though a driver might not use an optimal total route according to a given cost function, short parts of a route usually *are* optimal. Luxen and Schieferdecker (2012) combined the idea of locally optimal alternatives with an optimized road network that considers faster roads in the middle of a route. Paraskevopoulos and Zaroliagis (2013) introduced additional filtering methods to remove unwanted alternatives.

The following approach is heavily based on former research on route planning algorithms in den *HomeRun* environment (Roth 2013). The primary goal of former research was to provide algorithms *beyond* the typical task of optimal route planning. All of the following approaches are based on A* but reused the underlying structures in a novel manner:

- For a set of n starts and m targets: find all $n \cdot m$ optimal routes for any combination of start and target. This function is called *multirouting*. We presented an approach to provide multirouting in a single step that is far more efficient than executing $n \cdot m$ single route planning (Roth 2016).
- Supervising a real driven route up to a certain point: what is the area of *possible* targets, if we assume locally optimal driving? This is somehow the opposite question of route planning as we want to predict a locally optimal route from a partly supervised route. This function is called *target prediction*—we provide an efficient solution for it in (Roth 2014).
- Given a set of measured positions: what is the most probable route that approximates all positions—this is an extended version of the so-called *map matching problem* known from navigation systems. But in contrast to existing approaches, we consider all possible routes and take the one with the highest probability (Roth 2016b).

Based on the experience with unusual navigation questions, we wanted to solve the bypass problem. Even though strongly related to alternative path computation as described above, we can state certain differences. First, we strongly base on A* and thus can incorporate useful speed-up techniques, foremost the estimation based on ALT. Second, we do not limit the set of bypasses (e.g. only the shortest k) but return all. This allows an application or service to formulate any condition on its own and is not limited to any concept of alternative routes modeled by the algorithm. Even though 'all' instead of k routes seems to cause a high overhead, we present a structure that represents these routes, meanwhile it enables an efficient computation.

3 Computing Bypass Areas

3.1 Problem Statement

The concept of bypasses requires the knowledge of the optimal route that is implicitly defined by a *start*, *target* and *cost function* (e.g. driving time). We can formulate the bypass problem as follows:

> *What are all bypasses that do not exceed the minimal costs from start to target by a given factor?*

From these bypasses we want to know their paths, but also all passed positions (i.e. possible intermediate targets). Inevitable for any approach is a road graph that contains all crossings and connections between crossings. The latter specify road characteristics to compute costs, e.g. road type and speed limit.

More formally: 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*. Link costs $c(q_i, q_j)$ can be any positive number. 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 optimal route is a sequence of crossings (*start*, q_{i1}, q_{i2}, …, *target*) that minimizes the sum of link costs. We call the minimal costs between two (not necessarily connected) crossings c^*; in particular, the costs for our optimal route are $c^*(start, target)$.

3.2 Computing Optimal Paths

In order to present our bypass approach, we first describe A* that computes optimal single routes from a crossing to a crossing. In order to execute route planning in a target-oriented manner, A* requires a future path cost estimation function $h(q_i, target) \leq c^*(q_i, target)$ that provides a lower bound of costs for the route termination. We request h to be *monotone*, i.e. for two connected crossings q_i, q_j we get

$$\left|h(q_i, target) - h(q_j, target)\right| \leq c(q_i, q_j). \quad (1)$$

Important: also for bad estimations h, A* always produces optimal results. But the more h reaches the actual costs, the better A* performs, because it only deals with fewer unwanted crossings. Algorithm 1 presents the original A* approach as known from the literature.

Algorithm 1. Standard A*

```
A_star(start, target)
g[start]←0; f[start]←0; state[start]←open;
openList.add(start);                    // add start to open
for all q_i≠start {                     // initialize fields
    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;       // route to target found: finish
    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;   // the route start→…→q_i→q_j is less costly
                backLink[q_j]←q_j;            // than the formerly stored route
                state[q_j]←open;              // start→…→q_j (if any)
                openList.add(q_j);
            }
        }
    }
} while not openList.isEmpty();
return failure;                         // no route at all from start to target
```

Some remarks:

- We assign one of three states to each crossing: *not_visited*, *open* (g not finally computed), *closed* (optimal route from *start* discovered). The section '*expand crossing*' in the loop means: take an *open* crossing and check all its neighbors.
- $g[q_i]$ contains the currently *assumed* costs from *start* to crossing q_i. 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.
- $f[q_i]$ contains the currently *estimated* total costs from *start* to *target*, if a route goes through q_i.
- To efficiently get the open crossing with minimal f, we additionally need an *openList* that internally keeps the list sorted whenever an *open* crossing is added.
- 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*, also, if the crossing is

not part of the optimal route. We take advantage of this property in our approach (see next section).

A last important property of f that need in our approach: if we check the values of $f[q_i]$ in the do-loop, $f[q_i]$ cannot get smaller for a new iteration. In other words: if we expand a node q_i, none if its non-closed neighbors q_j can receive a smaller f-value. This is because

$$\begin{gathered} f[q_i] \leq f[q_j] \Leftrightarrow g[q_i] + h(q_i, target) \leq g[q_j] + h[q_j, target] \\ \Leftrightarrow h(q_i, target) \leq c(q_i, q_j) + h(q_j, target) \Leftrightarrow (q_i, target) - h(q_j, target) \leq c(q_i, q_j) \end{gathered} \tag{2}$$

which is true because h is monotone (formula (1)).

3.3 Bypass Areas—Basic Considerations

Once we are able to compute an optimal route between two crossings, we can think about alternative routes—our *bypasses*. Let $opt = c^*(start, target)$ denote the costs for the optimal route from *start* to *target* and $v \geq 1$ the *extension factor*. We are looking for all routes from *start* to *target* with costs of not more than $v \cdot opt$. As we are looking for the entire *bypass area*, we are actually looking for *all crossings* that are part of all such routes.

Another view to the problem (Fig. 1a): an *intermediate crossing I* of the network is part of the required bypass area, if

$$c^*(start, I) + c^*(I, target) \leq v \cdot opt. \tag{3}$$

Based on this consideration, we are able to provide a definition of a bypass area B:

$$B(start, target, v) = \{I | c^*(start, I) + c^*(I, target) \leq v \cdot c^*(start, target)\}. \tag{4}$$

Even though easy to formulate, a real implementation is not obvious. A naïve computation of B would iterate through all possible crossings of the road network with e.g. some million crossings. As the check, whether I belongs to B requires to compute c^* (and thus to execute A*) two times, this approach would require an unacceptable long computation. We thus present a solution that discovers all I more efficiently in the next section.

A further issue: even though the respective routes $start \rightarrow I \rightarrow target$ do not exceed the given cost limit, they sometimes do not meet the drivers expectation of an appropriate alternative route (Fig. 1b). The problem: the route can be split in two optimal subroutes, however the total route may lose optimality in the range of I. This contradicts the *local optimality* as described in Sect. 2. The example in Fig. 1b is a worst case of a *not* locally optimal route, as the subroutes to and from I contain reverse links and the driver has to U-turn at I.

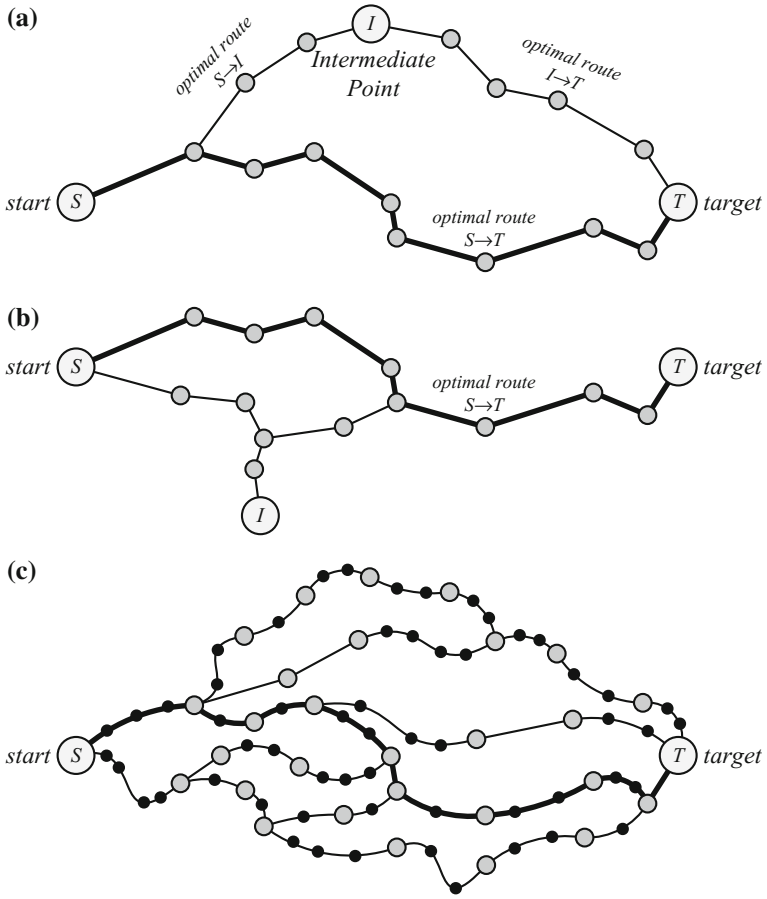

Fig. 1 Basic considerations about bypass area—**a** bypass through an intermediate point; **b** bypass without local optimality; **c** all bypasses with geometric representation

We thus introduce a second type of bypass area B_L that only contains intermediate crossings of routes with local optimality. It depends on the application scenario and user's expectation, whether local optimality is required (B_L) or not (B). To control the degree of local optimality we define a parameter ℓ with the meaning: all subroutes starting with a distance up to ℓ meters from I (on the route $start \to I$) and terminate up to ℓ meters to I (on the route $I \to target$) go through I. We can thus define B_L as follows:

$$B_L(start, target, v, \ell) = \{I | c^*(start, I) + c^*(I, target) \leq v \cdot c^*(start, target) \text{ and the route is locally optimal according to } \ell\}. \quad (5)$$

We provide an efficient approach to compute this set in the next section.

Figure 1c illustrates a last consideration: no matter whether we compute B or B_L, both sets only contain *crossings*. This is not sufficient if we want to compute an

area. As an example: if we want to detect appropriate hotels for a trip, we have to compute a geometric area that covers all line string points of corresponding routes, not only the crossings. Until now, all operations are based on the topological road network. For further operation, we have to shift to the geometric model.

3.4 Bypass Areas Without Local Optimality

We first want to compute the simple bypass area *B without* local optimality. To avoid iterating through all possible *I*, we take advantage of the way A* computes an optimal path: from the *start* crossing it iteratively considers new crossings and simultaneously maintains optimal routes from *start* to these crossings, even though they will not be part of the final route. A* terminates when it considers *target*, as the optimal route was found.

We modify the condition to terminate and we are able to create a field of all crossings *I* that *may* be element of *B*. As this field computes paths from the *start*, we call it *start field*. We additionally have to compute the second part of the route from *I* to *target*, represented by a *target field*. As each field is only able to consider its contribution to the condition in formula (4), each field contains more crossings than required. However, the set of crossings to consider is by far smaller than *all* crossings of the network.

Figure 2 illustrates the idea. The gray area presents all crossings that are *open* or *closed* and their *f*-values do not exceed $v \cdot opt$. All these crossings are reasonable candidates as from

$$c^*(start, q_i) + c^*(q_i, target) \leq v \cdot opt \tag{6}$$

and

$$f[q_i] = g[q_i] + h(q_i, target) = c^*(start, q_i) + h(q_i, target) \leq c^*(start, q_i) + c^*(q_i, target) \tag{7}$$

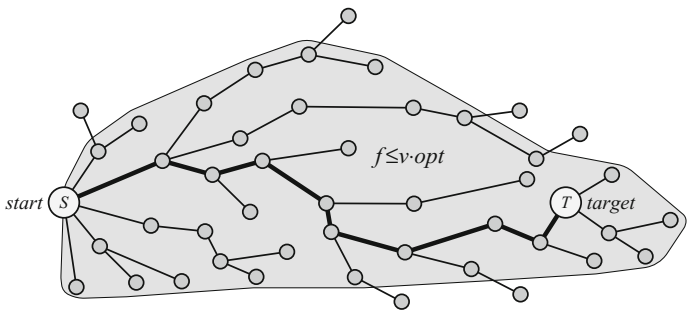

Fig. 2 Idea of start field generation

(which is true, because $h(q_i, target) \leq c^*(q_i, target)$) follows

$$f[q_i] \leq v \cdot opt. \tag{8}$$

As the *f*-value of the next crossing in the open list cannot get smaller (see formula (2)), we can stop when we poll a crossing with an *f*-value larger than $v \cdot opt$. As we do not know *opt* from the beginning, we first expand crossings as described in Algorithm 1 until we polled *target*. Then, we visit more crossings as far as we get the first crossing with $f > v \cdot opt$. As a consequence, the field goes beyond the *target*. This is reasonable, as those crossings are also candidates for *I*. Algorithm 2 sums up these considerations and shows how to compute the start field. To clearly distinguish the respective structures, we now apply index *s* to all arrays of the start field (e.g. g_s).

In a second round, we have to generate a target field with indexes *t* (Algorithm 3). The approach is similar to Algorithm 2, but in order to apply the appropriate driving direction, we have to incorporate these changes:

- The first *open* crossing is *target*.
- Whenever we expand a crossing q_i, we check the distance *from* the neighbor q_j, i.e. $c(q_j, q_i)$, not *to* the neighbor.
- The estimation is computed from *start* to the respective crossing.
- A sequence of *backLink* entries points to the *target* not to the *start*.

Algorithm 2. Start field generation

```
start_field(start, target, v)
g_s[start]←0; f_s[start]←0; state_s[start]←open;
openList.add(start);                    // add start to open
for all q_i≠start {                     // initialize fields
    g_s[q_i]← -1; f_s[q_i]←0; state_s[q_i]←not_visited;
}
opt←undefined;                          // costs for optimal route unknown for now
do {
    q_i←openList.poll();                // get q_i with state open and minimal f_s[q_i]
    if opt is defined and f_s[q_i]> v · opt  // this cannot be an intermediate crossing
        return success;                 // as the bypass would be too costly: finish
    if q_i=target opt←g_s[q_i];         // route to target found: set opt
    state_s[q_i]←closed;
    for all neighbors q_j of q_i {      // expand crossing
        …                               // see Algorithm 1
    }
} while not openList.isEmpty();
return failure;                         // no route at all from start to target
```

In addition, we can directly use the *opt* value of the start field. The target field generation can benefit from a much better estimation *h* compared to the start field

(see (*) in Algorithm 3): we set $h = g_s[q_j]$ whenever $g_s[q_j] \geq 0$. This does not change the result, but significantly reduces the number of visited crossings for the target field, thus improves the runtime. Using the g_s for h does not only provide an estimation, but returns the real costs, thus is the best considerable estimation.

The start and target field generation produce $g_s[q_i] = c^*(start, q_i)$ for all q_i with $state_s[q_i] = closed$ and $g_t[q_i] = c^*(q_i, target)$ for all q_i with $state_t[q_i] = closed$. As a consequence, we are now able to provide an efficient formula for B:

$$B(start, target, v) = \{q_i | state_s[q_i] = state_t[q_i] = closed \text{ and } g_s[q_i] + g_t[q_i] \leq v \cdot opt\} \tag{9}$$

This approach is by far more efficient than an approach that computes two optimal routes for *every* crossing in the network. The runtime of such a naïve approach would be million times longer than the single route planning. In contrast, our approach only causes a runtime of approx. 2 times more (see evaluation). Again note that in this time the information of *any bypass* in the road network with the required cost limit is discovered.

Algorithm 3. Target field generation

```
target_field(start, target, v)
g_t[target]←0; f_t[target]←0; state_t[target]←open;
openList.add(target);                    // add target to open
for all q_i≠target {                     // initialize fields
   g_t[q_i]← -1; f_t[q_i]←0; state_t[q_i]←not_visited;
}
do {                                     // Note: opt is known from start field
   q_i←openList.poll();                  // get q_i with state open and minimal f[q_i]
   if f_t[q_i]> v · opt                  // this cannot be an intermediate crossing
      return success;                    // as the bypass would be too costly: finish
   state_t[q_i]←closed;
   for all neighbors q_j of q_i {        // Note: driving direction is q_j→q_i !
      if state_t[q_j]≠closed {           // expand crossing
         g_new←g_t[q_i]+c(q_j, q_i);     // Note: costs from q_j→q_i !
         if g_s[q_j]≥0                   // (*) ideal estimation from start to q_j available
            h←g_s[q_j]                   // i.e. the real costs, taken from start field, g_s
         else
            h←h(start, q_j)              // not available: compute h yourself
         f_new←g_new+h;                  // Note: h estimates start→q_j !
         if state_t[q_j]=not_visited or f_new<f_t[q_j] {
            g_t[q_j]←g_new; f_t[q_j]←f_new;// the route q_j→q_i→…→target is less costly
            backLink_t[q_j]←q_j;         // than the formerly stored route
            state_t[q_j]←open;           // q_j→…→target (if any)
            openList.add(q_j);
         }
      }
   }
} while not openList.isEmpty();
return failure;                          // no route at all from start to target
```

3.5 Bypass Areas with Local Optimality

In a second step, we may reduce B to B_L that only considers locally optimal bypasses. Again note that this operation is application-dependent. In particular, we need to consider the user's expectation of a bypass.

Obviously $B_L \subseteq B$, thus we can iterate through all $I \in B$ and check, if the respective bypass is locally optimal. As the size of B can by large (e.g. some hundred thousand crossings), we must avoid multiple executions of A* to check the local optimality.

Some considerations: From a bypass through I, the parts $start \to I$ and $I \to target$ are already optimal, thus also *locally* optimal. The crucial part of the route is thus around I. If we require all subroutes up to length ℓ to be optimal, we can formulate this condition as follows:

- for a crossing q_i on the route $start \to I$ with a distance to I not more than ℓ, I is on the optimal route from $q_i \to target$ and
- for a crossing q_i on the route $I \to target$ with a distance to I not more than ℓ, I is on the optimal route from $start \to q_i$.

The distance ℓ can be expressed in any unit, e.g. meters for driving distances, but also in the same unit as c^*, e.g. the driving time. It depends on the application to choose an appropriate measure and value.

To avoid the costly check with the help of A* for every I we make use of an approach as illustrated in Fig. 3. We see the start and target field and a crossing $I \in B$.

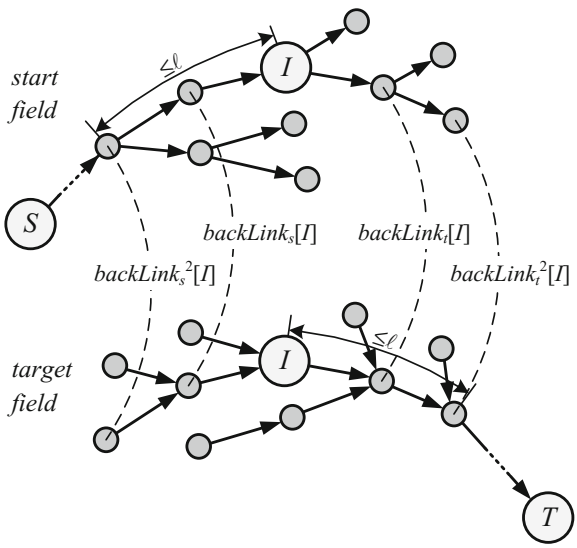

Fig. 3 Checking local optimality

Let $backLink^n[q_i]$ denote the n-times application of *backLink*, e.g., $backLink^2[q_i] = backLink[backLink[q_i]]$. We further define $backLink^0[q_i] = q_i$.

Obviously, $backLink_s^n[I] \in B$ for all n for which $backLink_s^n$ is defined. This is because

$$c^*\left(start, backLink_s^n[I]\right) + c^*\left(backLink_s^n[I], target\right) \leq c^*(start, I) + c^*(I, target). \tag{10}$$

As a consequence, $backLink_s^n[I]$ is also a *closed* crossing in the target field and its $backLink_t$ entry is defined.

On the subroute $start \rightarrow I$ we are interested in all crossings $backLink_s^n[I]$ that do not exceed distance ℓ to I. One of these crossings may violate the local optimality condition, i.e. the optimal route to *target* may not go over I. We thus get local optimality, if

$$backLink_t\left[backLink_s^n[I]\right] = backLink_s^{n-1}[I], \tag{11}$$

that can easily be checked for all n for which the corresponding entries are defined. In a second round we also have to check subroute $I \rightarrow target$ with corresponding crossings $backLink_t^n[I]$:

$$backLink_s\left[backLink_t^n[I]\right] = backLink_t^{n-1}[I]. \tag{12}$$

In the example in Fig. 3, the corresponding *backLink* entries are all equal, i.e. they all fulfil formulas (11) and (12). This proves the local optimality of the bypass over I. Algorithm 4 shows the final approach to compute B_L.

Algorithm 4. Computation of B_L

```
compute_BL(B, ℓ, backLink_s, backLink_t)
for all I∈B {                                   // main loop: check all I
    last=I; current=backLink_s[I];              // inner loop 1: I→start
    while distance between current and I not more than ℓ {
        if backlink_t[current]≠last             // route does not go over I
            remove I from B; continue main loop;  // I does not belong to B_L
        last=current; current=backLink_s[current];  // next in start direction
    }
    last=I; current=backLink_t[I];              // inner loop 2: I→target
    while distance between current and I not more than ℓ {
        if backlink_s[current]≠last             // route does not go over I
            remove I from B; continue main loop;  // I does not belong to B_L
        last=current; current=backLink_t[current];  // next in target direction
    }
}
return B;                                       // this set now is B_L
```

Some remarks about the runtime behavior: even though we have to loop over all $I \in B$, the two inner loops are not time-critical. This is because they only iterate up to a distance of ℓ. For an average link length, this is a constant, thus we have a complexity of O(|B|) to compute B_L.

3.6 Computing Geometric Areas, Samples

Once we computed our sets of bypass crossings B or B_L respectively, we now have to build structures that represent the bypass areas. We start with the definition of a route that goes through a bypass crossing q_i:

$$\begin{aligned} route(q_i) = (&start, backLink_s^{n-1}[q_i], \ldots, backLink_s^1[q_i], q_i, \\ &backLink_t^1[q_i], \ldots, backLink_t^{m-1}[q_i], target) \end{aligned} \tag{12}$$

whereas $start = backLink_s^n[q_i]$ and $target = backLink_t^m[q_i]$. Note that these routes are ordered sequences of crossings. We further define

$$routes(B^*) = \{route(q_i) | q_i \in B^*\} \tag{13}$$

for B^* that either is B or B_L. It represents all distinct routes through our bypass crossings. To switch to a geometric view, we further define $linestring(q_i, q_j)$ that represents the line string geometry of link (q_i, q_j). The *area* is the union of distinct line strings

$$area(B^*) = \{linestring(q_i, q_j) | (q_i, q_j) \text{ is link in } routes(B^*)\} \tag{14}$$

again for B^* that either is B or B_L.

Figure 4 presents the output of $area(B_L)$, i.e. the geometry of $routes(B_L)$, whereas B_L is computed for a given *start*, *target*, v and ℓ. As B_L only keeps intermediate points with local optimal driving, $routes(B_L)$ only contains reasonable *alternative routes*. Thus, we have an effective means to compute all alternative routes to a target that do not exceed a certain cost limit (10 % in Fig. 4). This is a useful service for all types of route planning. A navigation environment is able to immediately present the optimal routes as well as *all* reasonable alternatives on a map. The driver may decide to use a suboptimal route, e.g. to avoid a certain region or road. The navigation application could also provide a slider that allows a user to change v with immediate feedback of alternatives on the map.

We can also use the *area* function to query geo databases for interesting objects near this area. Figure 5 shows an example. It illustrates a search for gas stations near the route to the target, again with additional bypass costs that do not exceed 10 %. We use $area(B)$ here, as we accept to drive to and from the gas station *without* local optimality. Note that a real application would take into account the

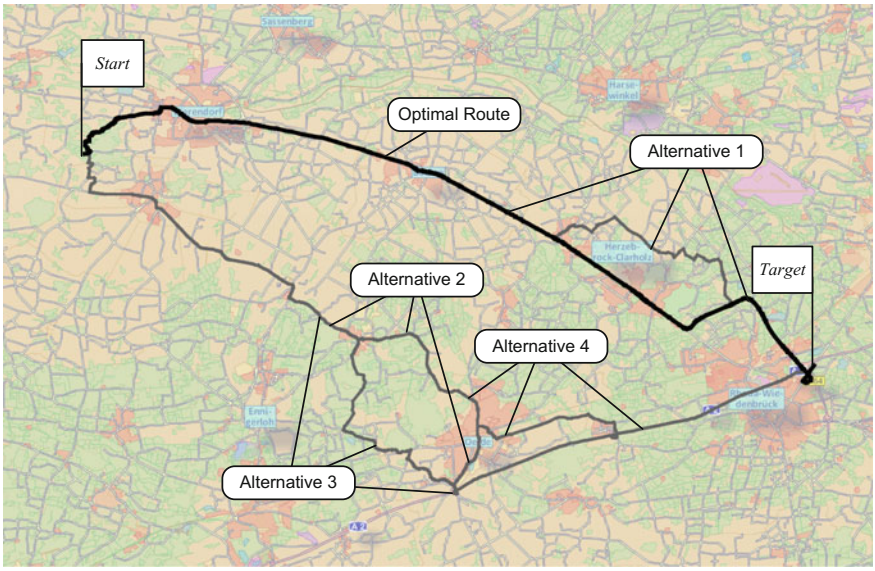

Fig. 4 Example computation of alternative routes (opt. route length = 35 km; $v = 1.1$; $\ell = 5000$ m)

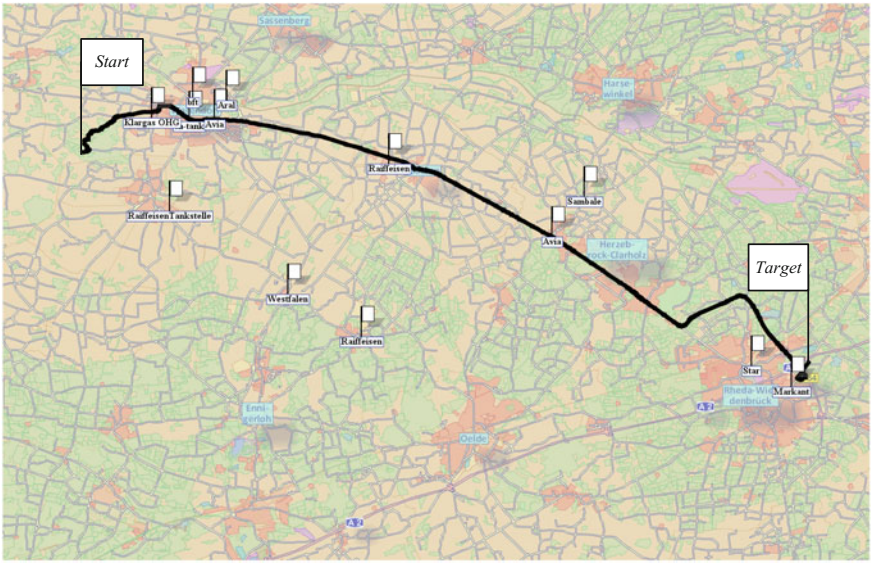

Fig. 5 Search for gas stations with limited bypass costs ($v = 1.1$; no local optimality)

current tank content to query for gas stations. If we are able to travel up to 80 % of the route with our tank content, we are only interested in gas stations that are 20–30 % from the target. This fine-tuning of queries, however, depends on the respective application.

We can additionally convert the set of line strings to a polygonal area using a *concave hull* operation. A concave hull is 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 α specifies how much the polygon geometrically adapts to the given coordinates. We use an approach based on the Delaunay Triangulation (Duckham et al. 2008). Figure 6 shows concave hulls with different values for v for our example.

Note that a concave hull also covers intermediate positions that do not belong to our bypass areas, as concave hulls do not create holes. On the other hand, all intermediate points that hold the cost limit have to be inside a hull. As a result, the concave hull region only provides an impression of reachable points, e.g. to be painted on a map. It also can be used to query a superset of results with a much simpler geometry than all line strings. This is reasonable as most spatial indexing mechanisms are based on a simplified geometry (e.g., Roth 2009) and an exact geometric check has to be performed in a second step.

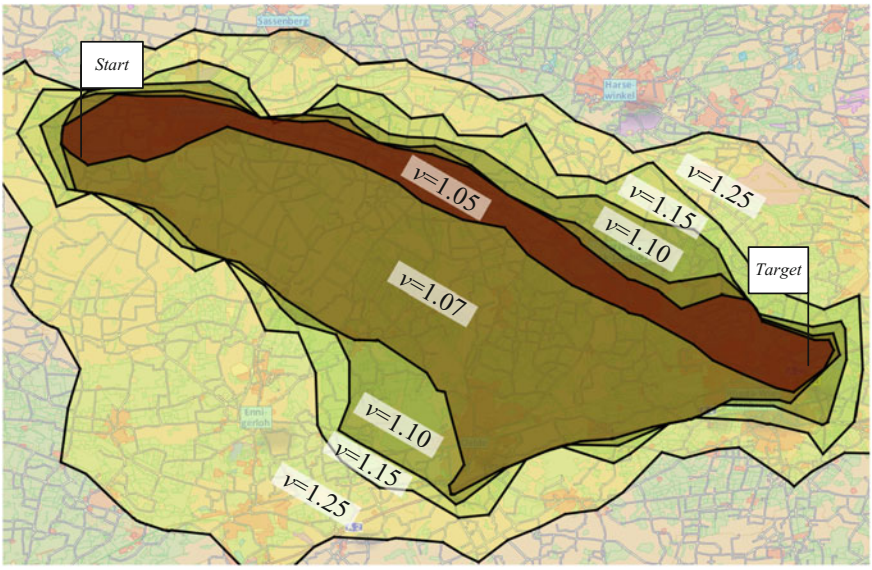

Fig. 6 Concave hulls of all bypass routes (parameter v; no local optimality)

4 Evaluation

We evaluated our implementation of the bypass functions in the HomeRun environment. The goal was to answer the following questions:

- For typical road networks and routes: what are the runtime costs to compute bypass areas?
- Does the approach to speed up the target field generation pays off?
- How costly is the detection of bypasses that are locally optimal?
- Finally: what are typical values for v and ℓ and what are their impact on the number of bypass crossings and alternative routes?

The following tests are based on the road network of Germany, imported from Open Street Map to our HomeRun navigation structure. The network contains 12.8 million crossings and 33.7 million links. For execution time measurements we used a PC with i7-4790 CPU, 3.6 GHz. We conducted 5 tests with different route lengths and values for v and ℓ. For each of these tests we selected 2000 random routes. Table 1 shows the results.

We first compared the time to compute the optimal route with the time to compute all bypasses (rows 4 and 5). The bypass computation takes 1.9–2.3 more computation time. This is a great result, as the bypass approach has to compute two fields, whereas we only need a single field to compute the optimal route—only this would require approx. twice the runtime of optimal routing. This proves that our approach to compute all bypasses is efficient.

Table 1 Bypass execution statistics (average values for 2000 executions per test)

	Test1	Test2	Test3	Test4	Test5
1: Distance start-target (km)	1–10	10–25	25–100	100–250	250–500
2: v	1.2	1.12	1.08	1.05	1.03
3: ℓ (km)	0.5	1.2	5	10	20
4: Execution time for opt. route (ms)	40	42	61	119	249
5: Bypasses execution time (ms)	77	83	134	282	587
6: Time to check local optimality (ms)	0.172	0.423	3.143	8.96	13.41
7: Visited crossings start field	4059	12 651	69 303	222 997	533 906
8: Visited crossings target field	1722	4127	21 013	66 633	145 651
9: Total bypass crossings	1425	3096	14 190	38 425	69 610
10: Ratio start field/bypass crossings	2.8	4.1	4.9	5.8	7.7
11: Ratio target field/bypass crossings	1.2	1.3	1.5	1.7	2.1
12: Bypass cross. with local optimality	306	516	1013	1471	1940
13: Alternative routes	18	17	16	12	6

Second, we checked, whether the speed up of the target field generation (marked by (*) in Algorithm 3) really pays off. Comparing rows 7 and 8 (also 10 and 11) we see the numbers of visited crossings of the target field are significantly lower than the numbers in the start field.

Third: we proved that the approach to detect locally optimal bypasses (Algorithm 4) is very efficient (row 6). The time compared to the total runtime can nearly be ignored.

Finally, we got an impression of bypasses (row 9), bypasses with local optimality (row 12) and alternative routes (row 13) in typical scenarios on real road network. The number of real alternatives is considerably low in our tests (less than 20 alternatives). This small list of routes could, e.g., be painted on a single map with different colors.

To sum up: the evaluation shows the effectiveness of the overall approach. It is possible with a considerable small execution time to compute the set of bypasses and, if required, to keep those with local optimality.

5 Conclusions

Our approach is an efficient solution for the bypass-problem: *what are the positions than can be reached with limited additional driving costs*? In contrast to similar questions (*k*-shortest paths, alternative routes), we are not limited to certain bypasses, but directly compute all within the cost limit. This enables applications to perform their own queries based on the bypass areas, such as: 'On the way to my target; which gas stations can be reached within 10 % additional driving costs?' And also: 'What are all alternative routes from start to target within 5 % more costs than optimal?'.

This approach is heavily based on A* and its structures. In particular, the approach benefits from the estimation function that does not affect the correct result but significantly speeds up the computation. The overall runtime is comparable to optimal path planning, thus a respective service could directly call the computation of bypass areas, whereas the optimal route can be considered as special type of bypass.

We fully implemented and evaluated our approach that is part of the donavio navigation environment in the HomeRun project.

References

Abraham I., Delling D., Goldberg A. V., Werneck R. F. (2010): Alternative Routes in Road Networks, in Proc. of the 9th Intern. Symposium on Experimental Algorithms (SEA '10).

Bellman R. and Kalaba R. (1960): On kth best policies, Journal of the Society for Industrial and Applied Mathematics, 582–588.

Dijkstra E. W. (1959): A note on two problems in connexion with graphs, Numerische Mathematik. 1, 1959, 269–271.

Duckham M., Kulik L., Worboys M. and Galton A. (2008): 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.

Eppstein D. (1994): Finding the k shortest paths, in Foundations of Computer Science, Proc of the 35th Annual Symposium on. IEEE, 154–165.

Geisberger R., Sanders P., Schultes D. and Delling D. (2008): Contraction Hierarchies: Faster and Simpler Hierarchical Routing in Road Networks, WEA 2008, LNCS 5038, 319–333.

Goldberg A. V., Harrelson C. (2005): Computing the Shortest Path: A* Search Meets Graph Theory, in Proc. 16th ACMSIAM, Symposium on Discrete Algorithms, pp. 156–165.

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

Hoffman W. and Pavley R. (1959): A method for the solution of the n th best path problem, Journal of the ACM (JACM), vol. 6, no. 4, 506–514.

Luxen D., Schieferdecker D. (2012): Candidate Sets for Alternative Routes in Road Networks, in Proc. of the 11th Intern. Symposium on Experimental Algorithms (SEA '12).

Paraskevopoulos A., Zaroliagis C. (2013): Improved Alternative Route Planning, 13th Workshop on Algorithmic Approaches for Transportation Modelling, Optimization, and Systems (ATMOS'13). 108–122.

Roth J. (2009): The Extended Split Index to Efficiently Store and Retrieve Spatial Data With Standard Databases, IADIS International Conference Applied Computing 2009, Rome, Nov. 19–21, 2009, Vol. I, 85-92.

Roth J. (2013): Combining Symbolic and Spatial Exploratory Search – the Homerun Explorer, Innovative Internet Computing Systems (I2CS), Hagen, June 19–21. 2013, VDI, Vol. 10, No. 826, 94–108.

Roth J. (2014): Predicting Route Targets Based on Optimality Considerations, International Conference on Innovations for Community Services (I4CS), Reims (France) June 4–6, 2014, IEEE xplore, 61-68.

Roth J. (2016): Efficient Many-to-Many Path Planning and the Traveling Salesman Problem on Road Networks, KES Journal: Innovation in Knowledge-Based and Intelligent Engineering Systems, 20 (2016), IOS Press, 135–148.

Roth J. (2016b): The Offline Map Matching Problem and its Efficient Solution, International Conference on Innovations for Community Services (I4CS), Vienna, under review.

Yen J. Y. (1971): Finding the k shortest loopless paths in a network, Management Science, vol. 17, no. 11, 712–716.