Acceleration of Shortest Path and Constrained Shortest Path Computation

Ekkehard Köhler[1], Rolf H. Möhring[1], and Heiko Schilling[1]*

TU Berlin, Germany,
{Ekkehard.Koehler|Rolf.Moehring|Heiko.Schilling}@TU-Berlin.DE

Abstract. We study acceleration methods for point-to-point shortest path and constrained shortest path computations in directed graphs, in particular in road and railroad networks. Our acceleration methods are allowed to use a preprocessing of the network data to create auxiliary information which is then used to speed-up shortest path queries. We focus on two methods based on Dijkstra's algorithm for shortest path computations and two methods based on a generalized version of Dijkstra for constrained shortest paths. The methods are compared with other acceleration techniques, most of them published only recently. We also look at appropriate combinations of different methods to find further improvements. For shortest path computations we investigate hierarchical multiway-separator and arc-flag approaches. The hierarchical multiway-separator approach divides the graph into regions along a multiway-separator and gathers information to improve the search for shortest paths that stretch over several regions. A new multiway-separator heuristic is presented which improves the hierarchical separator approach. The arc-flag approach divides the graph into regions and gathers information on whether an arc is on a shortest path into a given region. Both methods yield significant speed-ups of the plain Dijkstra's algorithm. The arc flag method combined with an appropriate partition and a bi-directed search achieves an average speed-up of up to 1,400 on large networks. This combination narrows down the search space of Dijkstra's algorithm to almost the size of the corresponding shortest path for long distance shortest path queries. For the constrained shortest path problem we show that goal-directed and bi-directed acceleration methods can be used both individually and in combination. The goal-directed search achieves the best speed-up factor of 110 for the constrained problem.

1 Introduction

In combinatorial optimization computing shortest paths is regarded as one of the most fundamental problems. What makes the shortest paths problem so interesting is the important role it plays in numerous real world problems: combinatorial models of real world scenarios often contain or reduce to shortest path computations. Much research has been done on shortest path problems and there is a large variety of different algorithms for computing shortest paths efficiently in a given network. In the present paper we look at one of the most common variants of the problem where one has to find a shortest path between two nodes in a directed graph, the *point-to-point shortest path problem* (*P2P*).

For a long time the main focus in developing shortest path algorithms has been on finding algorithms with good theoretical time-bounds. Probably the most famous and most frequently used P2P algorithm is the algorithm of Dijkstra [6] which runs in $O(m + n \log n)$ [10]. Although fast in theory, the corresponding algorithms are often not fast enough for applications in large networks that require very many shortest path computations. In our study we assume that for the same underlying network the shortest path problem has to be solved repeatedly. Thus, preprocessing of the network data is possible and can support the computations that follow.

The purpose of this paper is to study implementations of different acceleration methods for shortest path and constrained shortest path computations in traffic networks. In contrast to general graphs, road networks are very sparse graphs. They usually have an embedding in the plane and the considered arc lengths often resemble Euclidean distances.

For shortest path computations we compare several approaches, published only recently, and focus on four approaches which we found appropriate for our purpose. For shortest path computations we first

---

* Supported by the Deutsche Forschungsgemeinschaft (DFG) as part of the Research Cluster "Algorithms on Large and Complex Networks" (1126).

investigate a hierarchical multiway-separator method similar to Frederikson's [9]. In a preprocessing phase we determine a small node multiway-separator that divides the graph into regions of almost balanced size, then information is gathered on the distances between the separator nodes of that particular region and is used in subsequent shortest path computations. Second, we consider a generalization of a region-based arc labeling approach that we refer to as the *arc flag approach*. The basic idea of the arc flag method using a simple rectangular geographic partition has been suggested by Lauther [18]. The arc-flag approach divides the graph into regions and gathers information for each arc on whether this arc is on a shortest path into a given region. For each arc this information is stored in a vector. More precisely, the vector contains a flag for each region of the graph indicating whether this arc is on a shortest path into that particular region. Thus, the size of each vector is determined by the number of regions and the number of vectors is determined by the number of arcs. Arc flags are used in the Dijkstra computation to avoid exploring unnecessary paths.

In addition to ordinary shortest path computations, we also study accelerating constrained shortest path computations. We look at networks where each arc is assigned two values: a length and a cost. The aim is to compute a shortest path with respect to length such that the sum of the cost values of the corresponding arcs does not exceed a given cost bound. This is a well-known weakly NP-hard problem. A standard algorithm for constrained shortest paths is a generalized Dijkstra algorithm [2]. In the present paper we analyze the behavior of several acceleration methods for shortest paths when applied to the generalized Dijkstra algorithm. In particular we investigate a goal-directed search, a bi-directed search, and a combination of the two. The study is motivated by a routing project in cooperation with DaimlerChrysler AG. In this project we have to compute routes which guarantee a given fairness condition. This is where the constrained shortest path problem comes in.

To compare the different approaches we look at computational results for a given set of road and railroad networks. We first present the results for the main methods in this paper. We also compare them with results for other methods and finally we take into account combinations of our main methods with the other methods we discussed. In our tests we only present combinations which seem to be appropriate and leave out non-appropriate methods as, for example, pure goal-directed search in combination with the arc-flag method, since the latter is already highly goal-directed by construction.

Related work. There are various studies of acceleration methods for shortest path computations. A recent overview is given in [12]. There is a whole family of methods that use different kinds of hierarchies for the search process and make use of properties of the given graphs. A well known example by Frederikson [9] uses planarity of the input graph. More examples can be found in [1,3,4,26]. Gutman [14] introduces a method based on the concept of *reach*: for each node a single reach value together with Euclidean coordinates is stored in order to enable a specific kind of goal-directed search. Gutman reports that he computes shortest paths 10 times faster than the plain Dijkstra algorithm. Goldberg describes an approach which uses the A^* search in combination with a lower-bounding technique based on so-called *landmarks* and the triangle inequality. Using just one landmark the results that Goldberg's algorithm produces are not as good as Gutman's, but with 16 landmarks he reports on a speed-up of up to 17. The basic arc-flag approach using a rectangular geographic partition of the underlying graph is described by Lauther [19]. He observes an improvement by a factor of up to 64 compared to the plain Dijkstra algorithm. The partition used by Lauther is based on an embedding of the graph in the plane. Further experimental studies for geometric speed-up techniques can be found in Wagner and Willhalm [27], and for combinations of speed-up techniques in Holzer, Schulz, and Willhalm [15].

The constrained shortest path problem is NP-complete [11]. There are a number of papers studying dynamic programming approaches and approximation schemes, such as [17,25]. The labeling method for which we present acceleration techniques is described in [2]. Various enumeration methods can be used to tackle this problem, see e.g., [7]. Lagrangian relaxation of an ILP formulation of the constrained shortest path problem together with a gap closing technique is used in [20]. Recent experimental results can be found in [20,21,24].

Our contributions. For the hierarchical multiway-separator approach we extend Goodrich's algorithm to non-planar graphs. We introduce a new multiway-separator heuristic which for our data computes a smaller multiway-separator than Metis [22]. The sizes of the multiway-separator range between 67 % and 85 %

of the sizes of the separators computed by METIS. Because of the smaller multiway-separator size our heuristic improves the hierarchical multiway-separator approach by a speed-up factor of up to 14.

With the arc-flag method we investigate a new type of shortest path acceleration. It uses a partition of the node set of the graph into regions and precomputes one bit (flag) of information per arc and region. It consistently yields the best speed-up results on our road networks. When combined with an arc separator partition we obtain a speed-up factor of 220. A combination with a bi-directed search yields a speed-up factor of up to 1,400. It may seem promising to combine the arc flag method also with Gutman's acceleration [14]. However, our experiments have shown that although this method reduces the search space, it does not reduce the running time any further.

For the constrained shortest path problem we show that the goal-directed and the bi-directed approach can be used both individually and in combination. Here, the simple goal-directed search yields the greatest speed-up factors (110) and the bi-directed search which does not need preprocessing still provides reasonable results (factor of 5). To our knowledge this is the first time that these standard techniques have been applied to constrained shortest path problems on road networks.

Outline of the paper. We start in Section 2 with a description of the problem setting and our input networks. Section 3 introduces the two acceleration methods for shortest path: the hierarchical multiway-separator and the arc-flag method. In Section 4 we show the applicability of the standard acceleration techniques, goal-directed and bi-directed, to the constrained shortest path computation. The implementation of these methods is discussed in Section 5. Section 6 presents experimental results for all methods used in this study. We conclude with a detailed discussion of the results in Section 7.

2 Preliminaries

The input to the P2P problem is a directed graph $G = (V, A)$ with $n := |V|$ nodes, $m := |A|$ arcs, a source node s, a target node t and a nonnegative arc length $\ell(a)$, for each arc $a \in A$. Additionally, in the constrained shortest path case, there are nonnegative arc costs $c(a)$ for each arc $a \in A$.

The P2P problem is to find a length minimal path in a graph G from s to t, i.e., the sum of the arc lengths of all arcs in the path should be minimal. We will refer to the path as a *shortest* s, t*–path* in G and the sum of its arc length is denoted by $dist_s(t)$, the *shortest path distance* from s to t. In the *constrained P2P problem* the aim is to find a length minimal s, t–path for which the sum of the arc costs of all arcs in the path does not exceed a given *cost bound* $C_{s,t}$.

Our acceleration methods are based on Dijkstra's algorithm [6] which computes distance labels $d_s(u)$ from s to all reachable nodes $u \in V$ until $dist_s(t)$ is determined. The algorithm maintains a preliminary distance $d_s(v)$ for all nodes and a set S of nodes whose final shortest path distance from s has already been determined, i.e., $d_s(v) = dist_s(v)$. The algorithm starts with setting $d_s(s) = 0$ and inserts s into S. Then it repeatedly scans nodes $u \notin S$ in nondecreasing order of their distance label $d_s(u)$. It inserts u into S and updates labels of all adjacent nodes w with $(u, w) \in A$. Each node u is scanned and inserted into S at most once. On insertion arc (u, v) is considered and then $d_s(w)$ is updated by the sum $d_s(u) + \ell(u, w)$ if it is *dominated* by it, i.e., $d_s(u) + \ell(u, w) < d_s(w)$. Note, that it is not necessary for the algorithm to traverse the whole graph. The set of arcs which are traversed during the run of the algorithm is the *search space*. With our acceleration methods we restrict Dijkstra's algorithm to a smaller search space that still leads to the shortest path and thus results in a faster running time. Using auxilary precomputed information from our acceleration methods the algorithm is able to reject arcs before the update test which cannot be on a shortest path.

In the bi-directed search a second Dijkstra run is started simultaneously from t and computes a distance $dist_t(u)$ from t in the *reverse graph*, the graph with every arc reversed. The bi-directed search algorithm alternates between running the forward (common) and reverse search version of Dijkstra's algorithm and stops with an appropriate stopping criterion when the two searches meet. Note that any alternation strategy will correctly determine a shortest path.

In the constrained case we use a generalized version of Dijkstra's algorithm by Aneja, Aggarwal, and Nair [2]. Here, instead of one distance label per node a whole set of label pairs $(d_s(u), c_s(u))$ are used for each node u, each of them representing distance and cost of a path from s to u. If a node w is adjacent to u, all of its label pairs $(d_s(w), c_s(w))$ are removed if they are dominated by $(d_s(u)+\ell(u, w), c_s(u)+c(u, w))$, i.e., $d_s(u) + \ell(u, w) < d_s(w)$ and $c_s(u) + c(u, w) < c_s(w)$. Thus, the generalized version of Dijkstra's algorithm maintains a list of non-dominating label pairs at each node and stops once the target is reached.

3 Shortest Path Acceleration Methods

In this section we consider two acceleration methods for shortest path computation. Both methods have been used before for the case of planar embedded graphs. Here we extend them to work on almost planar graphs such as road or railroad networks. Note that in theory our extensions also work on arbitrary graphs.

3.1 The Multiway-Separator Approach

A significant acceleration can already be achieved by a divide and conquer method in combination with an appropriate preprocessing. In the *multiway-separator approach*, due to Frederickson [9], one computes a small node set whose removal partitions the graph into regions of roughly equal size such that there is no path between different regions.

Our heuristic to determine a balanced multiway-separator in road networks is based on an approach by Goodrich [13]. Goodrich uses multiway-separator that divides the graph in up to $O(n^\epsilon)$, $0 < \epsilon < \frac{1}{2}$, regions. The construction of the multiway-separator by Goodrich involves two steps. In the first step a breadth first search (BFS) tree from some root node s is computed and $O(n^\epsilon)$ so-called *starter-levels* ($0 < \epsilon < \frac{1}{2}$) in the tree are determined. For each of these starter-levels in the tree a *cut-level* above and below is determined, such that each cut-level contains maximally $2\lceil\sqrt{n}\,\rceil$ nodes and its distance is at most $\sqrt{n}/2$ levels away from the associated starter level. The nodes in the cut-level are marked as separator nodes. The size of the regions is bounded by $O(n^{1-\epsilon})$. In the second step Goodrich uses fundamental cycles to balance their size.

In our heuristic we make use of the first step of Goodrich's algorithm since the second step is not applicable to non-planar graphs. Instead, we apply again Goodrich's step 1 with a modified BFS-tree computation, together with a final cleaning step for merging small connected components. In that way we reduce the number of separator nodes and obtain regions of roughly equal size. Altogether, our multiway-separator heuristic consists of three steps: a BFS-tree computation for a coarse separation of the graph, a second BFS-tree computation for a finer separation, and a cleanup step. [1] The size of the resulting multiway-separator and the regions depend essentially on the choice of the different parameters in our multiway-separator heuristic, for example ϵ (used in step 1 and 2) and min_reg, max_reg as lower and upper bounds on the size of the regions in the cleanup step 3.

After the multiway-separator has been constructed, every node which is not in the multiway-separator is assigned to exactly one region. The separator nodes belong to all regions separated by them and are defined as border nodes for that region. Then all shortest paths between separator nodes of the same region are precomputed and the paths and their lengths are made available via lookup tables. This provides efficient access during the subsequent path searches.

Algorithm 1: Preprocessing for Hierarchical Multiway-Separator Acceleration Technique

```
Input  : directed graph $G = (V, A)$, nonnegative length $\ell_a$ for all $a \in A$, start and target nodes $s, t \in V$.
Output : multiway-separator partition, distance and path information between separator nodes.
1 find in $G$ a multiway-separator of small size and balanced region size
  using our multiway-separator heuristic ;
2 determine an all pair shortest path matrix for border nodes of each region
  together with path and distance information ;
```

For each determined shortest path between border nodes of a particular region an additional arc is introduced with the shortest path distance assigned to it as arc length. Border nodes together with the additionally inserted arcs form a *hierarchy layer* on top of the original graph. If a shortest path search starts

---

[1] Algorithm 5, appendix page 13.

at some node s lying in some region R_s the Dijkstra algorithm begins with scanning nodes in R_s. However, when leaving region R_s, the search algorithm walks only along arcs of the hierarchy layer, until it reaches the target region. Then, for the rest of the search it again walks along original arcs in the graph. If the determined shortest path streches over several regions it consists of original and additionally inserted arcs. But it can be reconstructed with the path information stored in the lookup table.

Our multiway-separator heuristic together with the hierarchical acceleration method for Dijkstra's algorithm delivers a speed-up factor of up to 14 compared to the plain Dijkstra's algorithm (see Section 6 for further results).

3.2 The Arc-Flag Approach

A significantly stronger speed-up can be achieved with the *arc-flag approach*. This approach is based on a partition of the graph into node sets $R_1, \ldots, R_k$, which we call *regions*. Each node is assigned to exactly one region. At each arc a we store a flag for each region R_i ($0 < i \leq k$). This flag is set to TRUE if a is on a shortest path to at least one node in R_i or if a lies in R_i, otherwise it is set to FALSE. For each arc a this information is stored in a vector of flags f_a. Thus the size of f_a is k, the number of regions, whereas the number of vectors is the number of arcs (see Figure 1). A shortest path search from a node s to a node t can now be conducted using a Dijkstra algorithm that only traverses arcs a with $f_a(j)$ is set to TRUE where R_j is the region containing vertex t.

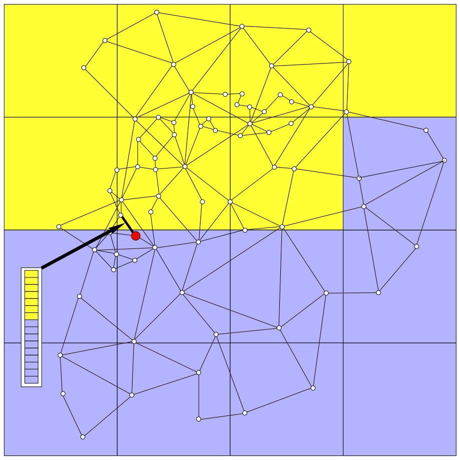

Fig. 1. In the arc-flag method at each arc a a vector f_a of arc-flags is stored such that $f_a[i]$ indicates if a is on a shortest path into region i.

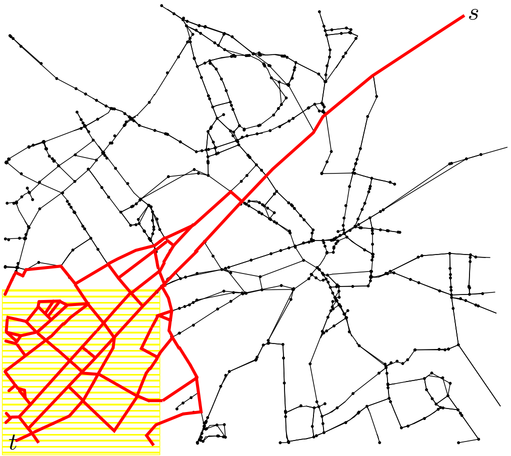

Fig. 2. The search space of a Dijkstra computation with arc-flag acceleration. The search started in s and the region containing the target node t is highlighted.

Algorithm 2: Preprocessing for the Arc-Flag Acceleration Technique

```
Input  : directed graph $G = (V, A)$, nonnegative length $\ell_a$ for all $a \in A$, start and target nodes $s, t \in V$.
Output : arc-flag vectors for each arc in $G$

cut the graph into regions ;
compute the vector of arc-flags for all arcs in the graph $G$ by repeated shortest path tree computations ;
```

The basic idea of this approach using a simple rectangular geographic partition has been reported by Lauther [18]. His partition requires an embedding of the graph in the plane. When combined with a bidirected search Lauther [19] obtains a speed-up factor of 64 on the European truck driver's road map (326,159 nodes, 513,567 arcs, 300 requests, 139 regions). With an improved partition of the graph we obtain a speed-up factor of 677 on an instance of roughly the same size (362,554 nodes, 920,464 arcs, 2,500 requests, 100 regions). Here we use fewer regions than Lauther and also apply bi-directed search. The speed-up of this method increases with larger instances up to a factor of 1,400 for our networks.

The preprocessing for this approach can be done as follows. Note that all shortest paths entering a region R_i have to use some arc a that crosses the border of R_i. Now for each of those crossing arcs a shortest path tree in the reverse graph is computed starting at arc a. All arcs in this a–rooted reverse shortest path tree obtain the value TRUE in their flag-vector at position i. Doing this for all arcs entering R_i one can fill up all entries at the i-th component of the flag-vector of all arcs in G. Note, that it is not possible to reduce the problem to one shortest path tree computation per region, since then it may be possible that we miss necessary flags. The set of arcs crossing the border of some region R form an arc cut $\mathcal{C}_R$. The total preprocessing time for that region then amounts to $O(|\mathcal{C}_R| n \log n)$. This can be reduced further since information on a computed shortest path tree of a border crossing arc a can be used for subsequent computations of shortest path trees of border crossing arcs of that region.

An additional reduction of the preprocessing time can be achieved by improving the partition in such a way that the total number of arcs crossing some regions border is reduced. This means to compute a small arc separator of our graph. Using METIS [22] for this task we can reduce the preprocessing time by a factor of 2 compared to the rectangular geographic partitioning, while using the same number of regions. Here the shortest path query time decreases by a factor of up to 4. The reason for this additional speedup is the fact that the arc separator determined by METIS much better represents the specific structure of the graph.

4 Constrained Shortest Path Acceleration Methods

In addition to accelerating shortest path queries we have also investigated the effect of standard acceleration methods for the resource constrained shortest path problem.

4.1 The Goal-Directed Approach

Our *goal-directed approach* attempts to accelerate the path search by employing a lower bound on the (remaining) path lengths and costs to the target node. This is achieved by modifying the arc lengths and costs and thereby forcing Dijkstra's algorithm to prefer nodes closer to the target node over those further away. In road and railroad networks usually Euclidean distances are used as lower bounds. However, in the case of constrained shortest paths one can exploit the fact that computing the shortest path from the target node to all other nodes is cheap compared to the overall cost of the extended Dijkstra. Hence, by simply computing both a reverse shortest path tree from the target node with respect to length and a reverse shortest path tree with respect to cost, it is possible to determine very good lower bounds on the remaining path lengths. One can use these trees for directing the constrained shortest path search. Note that here it is not necessary to restrict the method to Euclidean distances. Thus our goal-directed technique for constrained shortest paths is not limited to graphs that are embedded in the plane. Compared to other acceleration methods, this procedure consistently has delivered the best results in our tests (up to speed-up factors of 110).

More detailed, given lower bounds for lengths LBL, lower bounds for costs LBC, and a target node t, the following modified lengths $\ell'_{(v,w)}$ and costs $c'_{(v,w)}$ are used for arc (v, w):

$$(\ell'_{(v,w)}, c'_{(v,w)}) := (\, \ell_{(v,w)} - LBL_{(v,t)} + LBL_{(w,t)}, c_{(v,w)} - LBC_{(v,t)} + LBC_{(w,t)}) \,. \tag{1}$$

Since the extended Dijkstra algorithm works only for nonnegative arc costs the following consistency condition for the modified arc lengths and costs is vital to all types of lower bounds that we use:

$$\ell_{(v,w)} + LBL_{(w,t)} \geq LBL_{(v,t)} \qquad\qquad c_{(v,w)} + LBC_{(w,t)} \geq LBC_{(v,t)} \,. \tag{2}$$

For determining the lower bounds LBL and LBC we use the distances computed by plain Dijkstra from the target node t to all nodes u in G. This Dijkstra computation is done in the reverse graph; thus the consistency condition (2) is fulfilled.

If we apply the extended Dijkstra algorithm to the modified costs (1) we obtain a constrained shortest s, t–path $p = (s, v_1, \ldots, v_n, t)$ which does not exceed the original cost bound. For the length ℓ'_p of p we then obtain $\ell'_p = \ell'_{(s,v_1)} + \ell'_{(v_1,v_2)} + \ldots + \ell'_{(v_n,t)} = \ell_p - LBL_{(s,t)} + LBL_{(t,t)}$. Hence we have:

Proposition 1. *In the above setting, a feasible constrained shortest s,t–path with length ℓ'_p and costs c'_p determined by the extended Dijkstra algorithm using modified arc lengths and arc costs according to* (1) *is also a feasible constrained shortest path with length ℓ_p and costs c_p according to the original arc lengths and arc costs. Lengths and costs of the determined path fulfill $(\ell_p, c_p) = (\ell'_p + LBL_{(s,t)}, c'_p + LBC_{(s,t)})$.*

Algorithm 3: Goal-Directed Extended Dijkstra

```
Input  : directed graph $G = (V, A)$, nonnegative length and cost $(\ell_a, c_a)$ for all $a \in A$, start and target
         nodes $s, t \in V$, upper bound at the path costs $UBC$, $k$ (number of Pareto-optimal paths).
Output : up to $k$ shortest paths from $s$ to $t$ fulfilling the cost bound $UBC$.

determine lower bounds $LBL_{v,t}$ and $LBC_{v,t}$ ;
$(\ell'_{vw}, c'_{vw}) \leftarrow (\ell_{vw} - LBL_{v,t} + LBL_{w,t}, c_{vw} - LBC_{v,t} + LBC_{w,t}), \forall (v,w) \in A$ ;
$UBC' \leftarrow UBC - LBC_{(s,t)}$ ;
start extended Dijkstra algorithm with $(\ell'_a, c'_a)$ and $UBC'$ until $k$ paths are found ;
$(\ell_p, c_p) \leftarrow (\ell'_p + LBL_{(s,t)}, c'_p + LBC_{(s,t)})$ ;
```

4.2 The Bi-Directed Approach

In the *bi-directed approach* the constrained shortest paths are computed simultaneously from the start and the target node. In a traditional bi-directed search there is a simple stopping criterion to stop the search when the two frontiers meet. However, in the resource constrained case the stopping criterion is more complex. We use the following stopping criterion for the ordinary shortest path problem as a starting point for a generalization to constrained shortest paths.

Proposition 2. *Let $z := \min\{dist_s(v) + \ell_{(v,w)} + dist_t(w), v$ marked from s, w marked from t $\}$, $z_s := \min\{dist_s(u)$, u not marked from $s\}$, $z_t := \min\{dist_t(u)$, u not marked from $t\}$ and $z_0 := min\{z_s, z_t\}$. If $2 \cdot z_0 > z$, then the path with length z is a resource constrained shortest path for the given problem.*

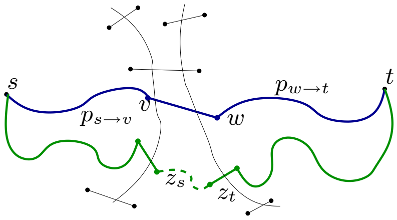

Fig. 3. Stopping criterion described in Proposition 2

The adaption of Proposition 2 requires the generalized Dijkstra to explore labels in a lexicographic order (length before cost). When in the situation of Figure 3 the upper path $p = p_{s \to v}, (v, w), p_{w \to t}$ is respects the cost bound, has currently shortest length and the value of z_0 with respect to length fulfills $2 \cdot z_0 > z$, then p is a constrained shortest path.

Remark. A combination of the goal-directed and the bi-directed approach can also be used for solving the constrained shortest path problem, but the stopping criterion in Proposition 2 is not optimal for the combined version. The lower bounds on the paths from the target node to the connecting node v do not cancel out in a telescopic sum, as they do in the goal-directed method in Section 4.1. We applied the stopping criterion of Proposition 3 (see appendix page 14), which slowed down the algorithm. So there might still be room for improvement if one finds a better stopping criterion.

5 Implementation

The methods presented in this paper were implemented in C++ using the GNU g++ compiler version 3.4.2 with the optimizing option ”-O3” on Linux 2.4/2.6 systems (SuSE 9.1). All computations were done on 64 bit machines: Intel Itanium II machines 1.2 GHz with 64 GB shared memory and 500 KB cache memory and AMD Opteron machines 2.2 GHz with 8 GB memory and 1 MB cache memory.

For this work we had to efficiently plug together and test several different versions of (constrained) shortest path algorithms. For developing such a framework we used generic programming techniques via template meta-programming in C++ as described, e.g., by Czarnecki and Eisenecker [5].

6 Experiments

6.1 Instances

All computations were done on real world data. In particular we used the networks in Table 1.

Table 1. Input networks used in the paper. Dijkstra running time is the shortest path query time computed by a plain Dijkstra's algorithm for a single request. The values are averaged over 2,500 requests which were computed on an Opteron processor (2.2 GHz).

name	description	# nodes	# arcs	Dijkstra running time [sec]
B	Berlin	12,100	19,570	0.1
GR	German Railway	14,938	32,520	0.2
GH	German Highway	53,315	109,540	1.0
AA	North Rhine-Westphalia south	362,554	920,464	5.3
TH	Thuringia	422,917	1,030,148	6.1
OS	Berlin, Brandenburg, Saxony, Saxony-Anhalt, Mecklenburg	474,431	1,169,224	7.0
NW	North Rhine-Westphalia north	560,865	1,410,076	9.9
NO	Lower Saxony, Schleswig-Holstein, Hamburg, Bremen	655,192	1,611,148	11.6
HS	Hesse, Saarland, Rhineland-Palatinate	675,465	1,696,054	11.7
BY	Bavaria	1,045,567	2,533,612	17.2

Each arc in these networks has a nonnegative integer geographic length. The arc costs are nonnegative rational numbers and arise from a routing project; see [16] for more details. For each network instance we randomly generated up to 2,500 route requests. The measured speed-up factors and running times are averaged over all computed requests.

6.2 Shortest Path Computations

Shortest path computations were done with the following acceleration methods and compared to Dijkstra's standard algorithm: multiway-separator heuristic (sep-heu), arc-flag approach together with a rectangular partition (af-rect) and an arc separator partition (af-sep). The arc separators were computed with METIS [22]. Combining the bi-directed search and the arc-flag method with an arc separator partition by METIS was the most successful method in our tests with a speed-up of up to 1,400 (af-sep-bi).

In all computations we measured the preprocessing time, the average over all shortest path requests of the shortest path query time, of the length (number of arcs) of the computed shortest path, and of the size of the shortest path search space. For the arc-flag method we also measured these parameters separately for the 10% shortest and the 10% longest requests with respect to their shortest path distance.

Table 2. Number of arcs of computed shortest path vs. search space on network OS (474,431 nodes, 1,169,224 arcs). The reduction of the search space is the fraction of the size of the search space vs. the number of arcs in the corresponding shortest path, averaged over all requests. |requests| is the number of computed requests, req.length is the relative length of the request, av. |s.path| is the average number of arcs of the determined shortest paths. Results are shown for the arc-flag method with a rectangular partition (af-sep) and a multiway-separator partition (af-sep) as well as a combination of these two with bi-directed search (af-rect+bidi, af-sep+bidi). All partitions consist of 225 regions. It is remarkable that the arc-flag method combined with a bi-directed search narrows the search space down to almost the size of the corresponding shortest path.

network	|requests|	req.length	av. |s.path|	plain Dijkstra	af-rect	af-sep	af-rect+bidi	af-sep+bidi
OS	250	long	791	× 1,313.3	× 11.0	× 6.9	× 1.3	× 1.2
	250	short	136	× 571.7	× 74.0	× 26.0	× 19.7	× 5.4
	2,500	all	437	× 1,342.8	× 23.0	× 11.0	× 2.4	× 1.8

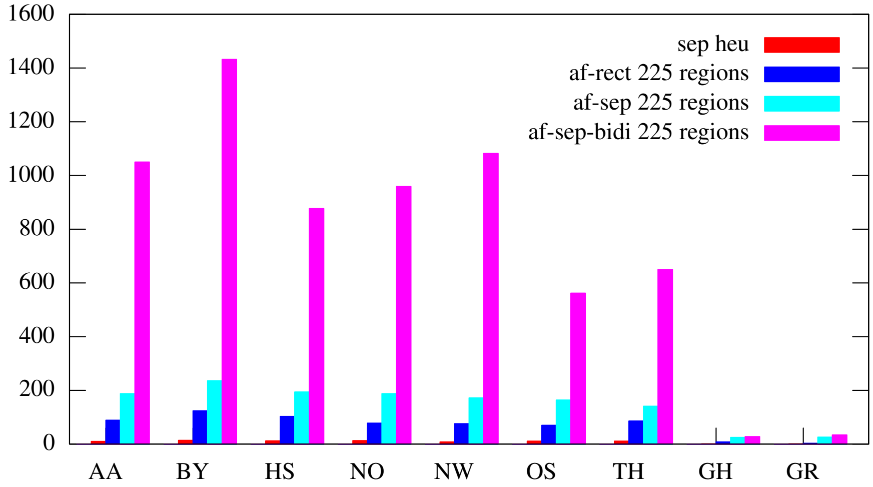

Fig. 4. Speed-up factors on all networks compared to the plain Dijkstra algorithms (factor of 1). Results are shown for the multiway-separator heuristic (sep-heu), the arc-flag with a rectangular partition (af-rect 225 regions), the arc-flag with a separator partition (af-sep 225 regions), and the arc-flag with a separator partition combined with bi-directed search (af-sep-bidi 225 regions).

Table 3. Speed-up factors of the arc-flag method with a multiway-separator partition with different numbers of regions and combined with bi-directed search. Speed-up factors are compared to the plain Dijkstra algortihm (factor of 1). Average values for 2,500 requests.

network	# nodes	# arcs	plain Dij.	25 regions	100 regions	225 regions	400 regions	625 regions
GR	14,938	32,520	× 1	× 8.56	× 18.4	× 32.6	× 38.39	× 43.7
GH	53,315	109,540	× 1	× 11.7	× 23.2	× 28.7	× 31.7	× 33.4
AA	362,554	920,464	× 1	× 260.6	× 677.6	× 1,020.5	× 1,136.9	× 148.9
OS	474,431	1,169,224	× 1	× 190.2	× 489.6	× 598.3	× 722.7	× 671.7

6.3 Constrained Shortest Path Computations

In our networks the path length corresponds to travel times and the cost bound to a geographic length bound on the paths. This is motivated by route guidance systems, where the cost corresponds to a given fairness

Table 4. Constrained shortest path acceleration methods on the Berlin road network (12,100 nodes, 19,570 arcs). Results are shown for the plain generalized Dijkstra (plain), the goal-directed search (go), the bi-directed search (bi), and the combination (go-bi). c. fact. is the constrained factor, av. |sh.path| is the average length of the determined shortest paths, |search space| is the size of the search space, max. |label list| is the maximum size of a node label list during a computation, prepro. time is the preprocessing time in seconds, comp. time is the computation time in seconds, and speed-up fact. is the speed-up factor compared to the plain generalized Dijkstra algorithm (factor of 1). computed values are on average for 1,000 requests, on Itanium II processor (1.2 GHz).

method	c. fact.	av. |sh.path|	|search space|	max. |label list|	prepro. time [s]	comp. time [s]	speed-up fact.
plain	1.2	68	80,803	23	0	1,212	× 1
go	1.2	68	1,484	24	161	13	× 93
bi	1.2	68	16,045	20	0	231	× 5
go-bi	1.2	68	18,401	39	339	197	× 6
plain	1.1	72	88,310	23	0	1,383	× 1
go	1.1	72	2,188	34	156	18	× 77
bi	1.1	72	21,942	22	0	350	× 4
go-bi	1.1	72	10,382	41	328	95	× 15
plain	1.05	76	95,241	23	0	1,534	× 1
go	1.05	76	1,847	26	154	14	× 110
bi	1.05	76	29,447	25	0	496	× 3
go-bi	1.05	76	4,793	26	323	37	× 42

condition. More precisely, the cost bound is determined by a factor times the geographic path length of an s, t–path with minimum length. In our experiments we tested factors of 1.05, 1.1 and 1.2. Computational results for constrained shortest path acceleration methods are displayed in Table 4.

7 Discussion and Conclusion

7.1 Shortest Path Acceleration Methods

The Multiway-Separator Approach. The technique from Section 3.1 achieved a speed-up factor of up to 14 compared to the plain Dijkstra algorithm, see Figure 4. Our multiway-separator heuristic was able to find a smaller multiway-separator on our road network instances than METIS (between 67 % and 85 % of the multiway-separator sizes computed by METIS). The heuristic was also able to improve upon the hierarchical separator methods by Frederickson [8]. But hierarchical separator methods together with our separator heuristic is not among the best acceleration techniques studied here. For our multiway-separator heuristic typical separator sizes are 1.67% (AA) and 1.34% (OS) of the graph nodes and the resulting regions are of balanced size. The preprocessing time of the hierarchical multiway-separator technique is comparably small, 32 min (AA) and 28 min (OS). The reason for the performance of the hierarchical approach is the number of artificially inserted arcs, which is 18.55% (AA) and 12.6% (OS) of the number of original graph arcs. This is not in line with the aim to reduce the search space in the shortest path computation. Another problem is the huge memory consumption of this method, which is higher than for all other methods we discussed: e.g., 4.5 GB on our largest instance BY. We also studied combinations of the hierarchical multiway-separator approach with both the goal-directed and the bi-directed search, but they only led to small improvements.

The arc-flag approach. For the arc-flag method there is a clear trade-off between speed-up factor and memory usage. Depending on the chosen partition, one can regard the arc-flag acceleration of shortest path computation as an interpolation between no precomputed information at all (plain Dijkstra) and complete precomputation by determining all possible shortest paths of the graph. Whereas the former is achieved by choosing a partition of the graph into just one region, the latter means partitioning the graph in such a way that a region is given for every single node of the graph. Thus in theory we can get as close as possible to the ideal shortest path search by increasing the number of regions in the partition ('ideal' means that the shortest path algorithm visits only arcs which actually belong to the shortest path itself). Obviously, an increase in the number of regions also entails an increase in preprocessing time and memory consumption (e.g., 625 regions for AA: 5.3 h preprocessing time; 1.6 GB memory).

Using a combination of this method together with other techniques, the best result that we achieved was on the largest instance (BY). The arc flag together with an arc separator partition and combined with a bi-directed search delivers a speed-up factor of 1,400 compared to the plain Dijkstra. In this case we used a partition into 225 regions. Thus we need an extra space of 450 bits ($\approx$ 56 byte) per arc.With just 6 bytes of information per arc (25 regions) the arc-flag method together with a bi-directed search consistently delivered speed-up factors of up to 260 (instance AA, table 3). This was also the best result for memory consumption vs. speed-up factor. Moreover, table 3 shows that this method is suitable in particular for larger instances (OS, AA) and long distance requests. For long requests on OS we narrowed the search space down to a factor of 1.2 times the number of shortest path arcs (corresponding to a speed-up factor of 844), while we consistently achieved a factor of around 1.7 for a partition with 225 regions (corresponding to a speed-up factor of 598).

Another important point is the relatively small preprocessing time for our big instances when compared to other preprocessing methods. For example, using the separator partition with 100 regions took us about 2.5 hours (OS) or 2.9 hours (AA). This is about half of what was spent on the rectangular partition. Still, the overall preprocessing time can be reduced even further by using information on shortest path trees which have been computed already for a region during the preprocessing phase.

Although we used this method on road networks with a given embedding in the plane, the question of whether one really needs such an embedding of the graph depends on the partition method that is used. Obviously, for the rectangular partition it is needed, but for a arc separator partition it is not. The choice of the underlying partition is crucial for the speed-up of this method. Using an arc separator partition instead of the rectangular partition results in an additional speed-up factor of 4. The reason for this is the fact that the arc separator partition determined by the help of METIS adapts much better adapts to the specific structure of the graph under consideration.

As for combinations of acceleration methods, the bi-directed search seems to be a perfect match for the arc-flags (additional speed-up factors of up to 7). The goal-directed search is less useful for the arc-flag method, since the method is already goal-directed by construction. Typically, the arc-flag method creates a cone-like spreading of the search space as it approaches the target region. In fact, at the beginning of a shortest path search the algorithm is forced by the arc-flags to walk along shortest path arcs only. Just before the target region is reached we can observe the spreading that was described above (see Figure 2). To cope with this behavior of the arc-flag method and in order to improve the algorithms even further, we suggest to study 2-level partitions; a coarse partition for far away target nodes and a finer one for nearby nodes. An extensive investigation of the speedup that can be obtained along these lines is presented in [23].

7.2 Constrained Shortest Path Acceleration Methods

In Section 4 we explained how to adapt well-known acceleration techniques for shortest path computations to the constrained shortest path search. Here the goal-directed search yields the best results, but the bi-directed search still delivers good accelerations. The advantage of the bi-directed search is that it does not require any additional preprocessing. The combined version (goal- and bi-directed) suffers from the lack of a good stopping criterion. See Table 4 for computational results of these methods.

The preprocessing phase of our methods is comparably short: on the Berlin road network for the goal-directed search the preprocessing takes up to 161 seconds and for the combined version up to 339 seconds. In the combined case the preprocessing requires two shortest path tree computations to compute the lower bounds from the target node to all other nodes in the graph. On hard instances where a large number of labels is created during the search process, the preprocessing time can be neglected compared to the overall processing time of a non-accelerated generalized Dijkstra run. A further advantage of the goal-directed search is the possibility to have more than one Pareto-optimal path computed within one run. This is of particular importance for applications such as the routing project mentioned before.

Acknowledgment. We would like to thank Benjamin Wardemann and Ronald Wotzlaw for their help with the implementation and the group of Dorothea Wagner at the University of Karlsruhe and the group of Kai Nagel at ETH Zurich for providing access to Opteron machines.

References

[1] R. Agrawal and H. Jagadish, *Algorithms for searching massive graphs*, IEEE transactions on knowledge and data engeneering, 6 (1994), pp. 225–238.

[2] Y. P. Aneja, V. Aggarwal, and K. P. K. Nair, *Shortest chain subject to side constraints*, Networks, 13 (1983), pp. 295–302.

[3] A. Car and A. Frank, *Modelling a hierachy of space applied to large road networks*, in Proceedings of IGIS'94, International Workshop on Advanced Research in Geographic Information Systems, 1994, pp. 15–24.

[4] Y.-L. Chou, H. E. Romeijn, and R. L. Smith, *Approximating shortest paths in large-scale networks with an application to intelligent transportation systems*, INFORMS Journal on Computing, 10 (1998), pp. 163–179.

[5] K. Czarnecki and U. W. Eisenecker, *Generative programming: methods, tools, and applications*, ACM Press/Addison-Wesley Publishing Co., 2000.

[6] E. W. Dijkstra, *A note on two problems in connexion with graphs*, Numer. Mathematik, (1955), pp. 269–271.

[7] D. Eppstein, *Finding the k shortest paths*, SIAM Journal on Computing, 28 (1999), pp. 652–673.

[8] G. N. Frederickson, *Fast algorithms for shortest paths in planar graphs, with applications*, SIAM J. Comput., 16 (1987), pp. 1004–1022.

[9] G. N. Frederikson, *Fast algorithms for shortest paths in planar graphs, with applications*, SIAM Journal on Computing, 16 (1987), pp. 1004–1022.

[10] M. L. Fredman and R. E. Tarjan, *Fibonacci heaps and their uses in improved network optimization algorithms*, J. ACM, 34 (1987), pp. 596–615.

[11] M. Garey and D. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*, W.H. Freeman, New York, 1979.

[12] A. V. Goldberg and C. Harrelson, *Computing the shortest path: A^* search meets graph theory*, Tech. Report MSR-TR-2004-24, Microsoft Research, 2003. Accepted at SODA 2005.

[13] M. T. Goodrich, *Planar Separators and Parallel Polygon Triangulation*, Journal of Computer and System Sciences, 51 (1995), pp. 374–389.

[14] R. Gutman, *Reach-based routing: A new approach to shortest path algorithms optimized for road networks*, in Proc. 6th Workshop on Algorithm Engineering and Experiments (ALENEX), 2004.

[15] M. Holzer, F. Schulz, and T. Willhalm, *Combining speed-up techniques for shortest-path computations*, in Experimental and Efficient Algorithms: Third International Workshop, (WEA 2004), C. C. Ribeiro and S. L. Martins, eds., vol. 3059 of LNCS, Springer, 2004, pp. 269–284.

[16] O. Jahn, R. H. Möhring, A. S. Schulz, and N. E. S. Moses, *System optimal routing of traffic flows with user constraints in networks with congestion*, Tech. Report 754, Technische Universität Berlin, 2002. to appear in Operations Research.

[17] H. Joksch, *The shortest route problem with constraints*, Journal of Mathematical Analysis and Application, 14 (1966), pp. 191–197.

[18] U. Lauther, *Slow preprocessing of graphs for extremely fast shortest path calculations*, 1997. Lecture at the Workshop on Computational Integer Programming at ZIB.

[19] ———, *An extremely fast, exact algorithm for finding shortest paths in static networks with geographical background*, in Geoinformation und Mobilität - von der Forschung zur praktischen Anwendung, Beiträge zu den Münsteraner GI-Tagen 2004 01./02. Juli 2004, M. Raubal, A. Sliwinski, and W. Kuhn, eds., vol. 22 of IfGI prints, Institut für Geoinformatik, Münster, 2004, pp. 22–32.

[20] K. Mehlhorn and M. Ziegelmann, *Resource constrained shortest paths*, in Proc. 8th European Symposium on Algorithms (ESA2000), LNCS 1879, Springer, 2000, pp. 326–337.

[21] ———, *CNOP - A package for constrained network optimization*, in 3rd Workshop on Algorithm Engineering and Experiments (ALENEX01), LNCS 2153, Springer, 2001, pp. 17–31.

[22] METIS, *A family of multilevel partitioning algorithms*. http://www-users.cs.umn.edu/∼karypis/metis.

[23] R. H. Möhring, H. Schilling, B. Schütz, D. Wagner, and T. Willhalm, *Partitioning graphs to speed up Dijkstra's algorithm*, (2005). Submitted to WEA 2005.

[24] M. Müller-Hannemann and M. Schnee, *Finding all attractive train connections by multi-criteria pareto search*, in 4th Workshop on Algorithmic Methods and Models for Optimization of Railways (ATMOS), 2004.

[25] C. A. Phillips, *The network inhibition problem*, in Proceedings of the Twenty-Fifth Annual ACM Symposium on the Theory of Computing, 1993, pp. 776–785.

[26] F. Schulz, D. Wagner, and C. Zaroliagis, *Using multi-level graphs for timetable information in railway systems*, in Proceedings 4th Workshop on Algorithm Engineering and Experiments (ALENEX), vol. 2409 of LNCS, Springer, 2002, pp. 43–59.

[27] D. Wagner and T. Willhalm, *Geometric speed-up techniques for finding shortest paths in large sparse graphs*, in Proc. 11th European Symposium on Algorithms (ESA 2003), G. D. Battista and U. Zwick, eds., vol. 2832 of LNCS, Springer, 2003, pp. 776–787.

A Acceleration Methods

A.1 Shortest Path Acceleration Methods

Algorithm 4: Modified breadth first search for multiway-separator construction

```
Input  : directed graph $G = (V, A)$, start-node $s \in V$
Output : partial-bfs-tree with root $s$
$level(s) := 0,\ level(v) := \infty\ \forall v \in V \setminus \{s\},\ R := \emptyset,\ Q := \{s\},\ pred(v) := NULL\ \forall v \in V$;
while $Q \neq \emptyset$ do
    choose $v \in Q$ with smallest $level(v)$;
    $Q := Q \setminus \{v\}$; $R := R \cup \{v\}$;
    if $v$ is no multiway-separator node then
        foreach $e := (v, w) \in \delta(v)$ do
            if $w \notin R$ then
                if $level(w) > level(v) + 1$ then
                    $level(w) := level(v) + 1$;
                    $pred(w) := v$;
                if $w \notin Q$ then $Q := Q \cup \{w\}$
```

Algorithm 5: Construction of a multiway-separator in a road network

```
   Input  : directed graph $G = (V, A)$, start-node $s$
   Output : set $S$ of separator nodes in $G$.
1  set $\epsilon \in (0; 1/2]$;
2  construct a BFS-tree with root $s$ in $\widetilde{G}$;
3  compute number of nodes $L(\ell_i)$ for each level $\ell_i$ and sum of nodes $s_{\ell_i}$ from level 0 to level $\ell_i$;
4  determine starter-level such that $(s_{\ell_i - 1}, s_{\ell_i}]$ contains a multiple of $\lceil n^{1-\epsilon} \rceil$;
5  foreach starter-Level $\ell_j$ do
       determine densest cut-level $\ell_{j_1},\ \ell_{j_2}$ such that $\ell_{j_1},\ \ell_{j_2}$ contain maximal $2\lceil\sqrt{n}\rceil$ nodes;
6  mark all nodes of cut-level as separator nodes;
7  mark all separator nodes as visited and all other nodes as unvisited;
   while an unvisited node $v$ exists do
8      construct a modified BFS-tree $T$, $n := |T|$, using algorithm 4 with arbitrary unvisited root $v$ in $\widetilde{G}$;
9      mark all nodes of $T$ as visited;
10     set $\epsilon \in (0; 1/2]$;
11     repeat steps 3 to 6 for the determined tree;
12 mark all non-separator nodes as unvisited;
13 set $0 \leq min\_reg < max\_reg \leq |V|$;
   while an unvisited node $v$ exists do
14     repeat steps 8 and 3;
       if $n \leq max\_reg$ then
15         mark all nodes of $T$ as visited
       else
16         let $\ell_i := \ell_j$ with $s_{\ell_j - 1} < min\_reg \leq s_{\ell_j}$;
17         if $\ell_j$ exists with $min\_reg \leq s_{\ell_j} \leq max\_reg$ then
               set $\ell_i := \min_{\ell_j}\{L(\ell_j) \mid min\_reg \leq s_{\ell_j} \leq max\_reg\}$;
18         mark all nodes of level $\ell_i$ as separator nodes;
19         mark all nodes of level $\leq \ell_i$ as visited;
```

Separator Approach Algorithm 5 describes the modified construction of a multiway-separators in more detail. The steps 12 to 19 represent the extension suggested in 3.1 for the partition of too large regions.

The sizes of the resulting multiway-separator and the determined regions essentially depend on the choice of the different parameters (steps 1, 10 and 13 of the algorithm 5).

A.2 Constrained Shortest Path Acceleration Methods

The Goal-Directed and Bi-Directed Approach

Proposition 3. *Let $best\_val$ be the original length and cost pair of the lexicographical smallest s, t–path regarding to the original lengths and costs, which was found so far by the goal-directed and bi-directed approach. p is optimal if the following inequality holds for the smallest unmarked list entry d'_min:*

$$d'\_min \geq_{lex} best\_val - (LBL(s,t), LBC(s,t))\,.$$

B Experiments

B.1 Shortest Path Computations

Table 5. Speed-up factors of the arc-flag method (af) compared to plain Dijkstra (factor of 1) on different networks. Results are shown for different numbers of regions in a rectangular partition (af-rect) and a separator partition (af-sep). Furthermore, the combination with bi-directed search (+bidi) is tested. Values are averaged over 2,500 computed requests.

network	regions	plain Dij.	af-rect	af-sep	af-rect+bidi	af-sep+bidi
OS	25	× 1	× 8.6	× 24.9	× 44.8	× 190.2
	100	× 1	× 35.9	× 82.5	× 269.7	× 489.6
	225	× 1	× 68.9	× 158.5	× 407.9	× 598.3
	400	× 1	× 98.7	× 229.6	× 520.7	× 722.7
	625	× 1	× 126.3	× 308.9	× 618.8	× 671.7
GH	25	× 1	× 3.8	× 7.2	× 7.4	× 11.7
	100	× 1	× 9.6	× 16.8	× 17.4	× 23.2
	225	× 1	× 13.8	× 23.4	× 24.8	× 28.7
	400	× 1	× 16.1	× 27.9	× 27.5	× 31.7
	625	× 1	× 18.4	× 31.3	× 29.7	× 33.4
GR	25	× 1	× 1.24	× 6.4	× 1.8	× 8.56
	100	× 1	× 2.8	× 15.4	× 5.2	× 18.4
	225	× 1	× 3.4	× 27.5	× 7.7	× 32.6
	400	× 1	× 5.1	× 35.6	× 11.9	× 38.39
	625	× 1	× 6.8	× 44.9	× 66.6	× 43.7

Table 6. Speed-up factors on different networks compared to plain Dijkstra (factor of 1). Results are shown for the reach method by Gutman [14] (reach), the arc-flag method with a separator partition into 25 regions (af-sep), the arc-flag with a separator partition into 25 regions combined with the reach method (af-sep+reach), the arc-flag with a separator partition into 25 regions combined with a bi-directed search (af-sep+bidi); the arc-flag with a separator partition into 25 regions combined with the reach method and bi-directed search (af-sep+reach+bidi); computed on an Opteron processor (2.2 GHz).

instance	# nodes	# arcs	basic	reach	af-sep	reach+af-sep	af-sep+bidi	af-sep+reach+bidi
B	10,491	17,038	× 1	× 1.4	× 6.5	× 7.0	× 11.6	× 12.4
B1	20,741	59,760	× 1	× 1.8	× 10.6	× 13.1	× 32.9	× 28.0
B2	19,411	55,536	× 1	× 2.0	× 10.6	× 13.7	× 31.6	× 25.6
B3	19,925	57,196	× 1	× 1.9	× 11.1	× 14.0	× 33.7	× 32

Table 7. Separator sizes computed with our multiway-separator heuristic and with METIS on all networks. #regions is the determined number of regions, #separator nodes (sep heu) is the determined separator size computed by our separator heuristic, and #separator nodes (METIS) is the determined separator size computed by METIS.

instance	#nodes	#arcs	#regions	#separator nodes	
				sep heu	METIS
AA	362,554	920,464	174	6,043	7,248
BY	1,045,567	2,533,612	463	13,390	16,226
HS	675,465	1,696,054	373	9,490	11,173
NO	655,192	1,611,148	392	9,797	12,510
NW	560,865	1,410,076	260	9,911	11,288
OS	474,431	1,169,224	411	6,362	9,447
TH	422,917	1,030,148	335	5,953	8,265

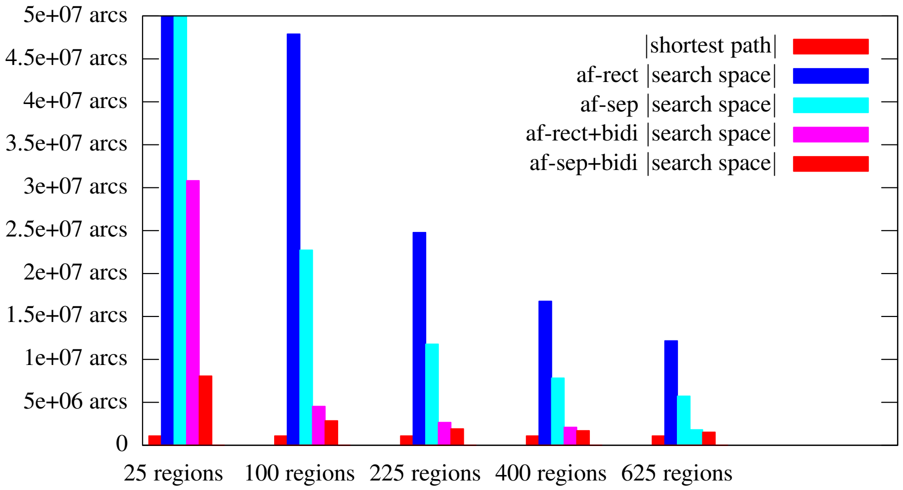

Fig. 5. Search space sizes of the arc-flag method on instance OS (474,431 nodes, 1,169,224 arcs). The y-axis shows the size of the search space, while the x-axis shows different partition sizes in terms of numbers of determined regions. |shortest path| is the average number of arcs of all determined shortest paths. Based on different sizes of the underlying partition, results are shown for the arc-flag method with a rectangular partition (af-rect), the arc-flag method with an arc separator partition (af-sep), the arc-flag method with a rectangular partition combined with bi-directed search (af-rect+bidi), and the arc-flag method with an arc separator partition combined with bi-directed search (af-sep+bidi).

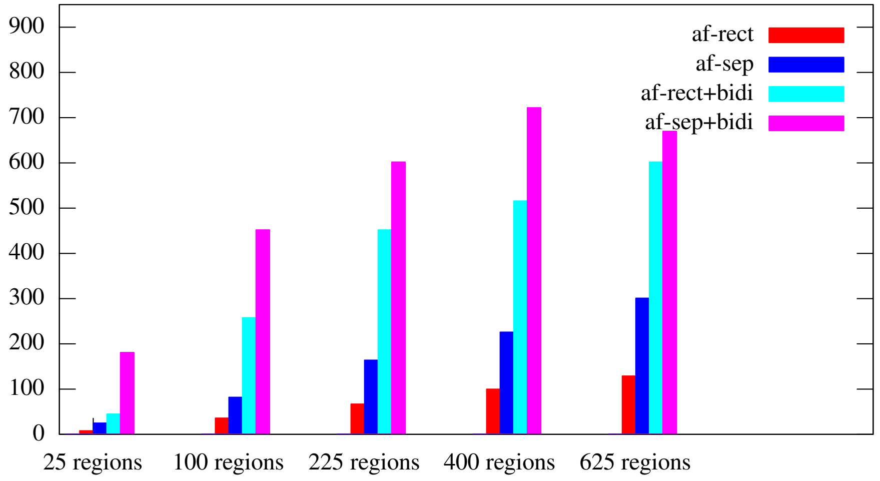

Fig. 6. Speed-up factors of the arc-flag method on instance OS (474,431 nodes, 1,169,224 arcs) compared to plain Dijkstra (factor of 1). The y-axis shows the size of the achieved speed-up factors, while the x-axis shows different partition sizes in terms of numbers of determined regions. Based on different sizes of the underlying partition, results are shown for the arc-flag method with a rectangular partition (af-rect), the arc-flag method with an arc separator partition (af-sep), the arc-flag method with a rectangular partition combined with bi-directed search (af-rect+bidi), and the arc-flag method with an arc separator partition combined with bi-directed search (af-sep+bidi).

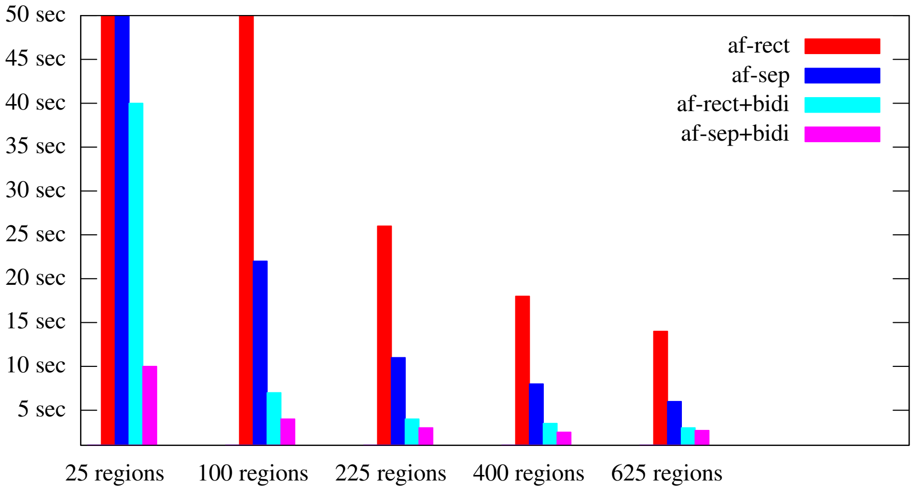

Fig. 7. Running times in seconds of the arc-flag method on instance OS (474,431 nodes, 1,169,224 arcs). The y-axis shows the running time in seconds of a shortest path query average over all requests, while the x-axis shows different partition sizes in terms of numbers of determined regions. Based on different sizes of the underlying partition, results are shown for the arc-flag method with a rectangular partition (af-rect), the arc-flag method with an arc separator partition (af-sep), the arc-flag method with a rectangular partition combined with bi-directed search (af-rect+bidi), and the arc-flag method with an arc separator partition combined with bi-directed search (af-sep+bidi).

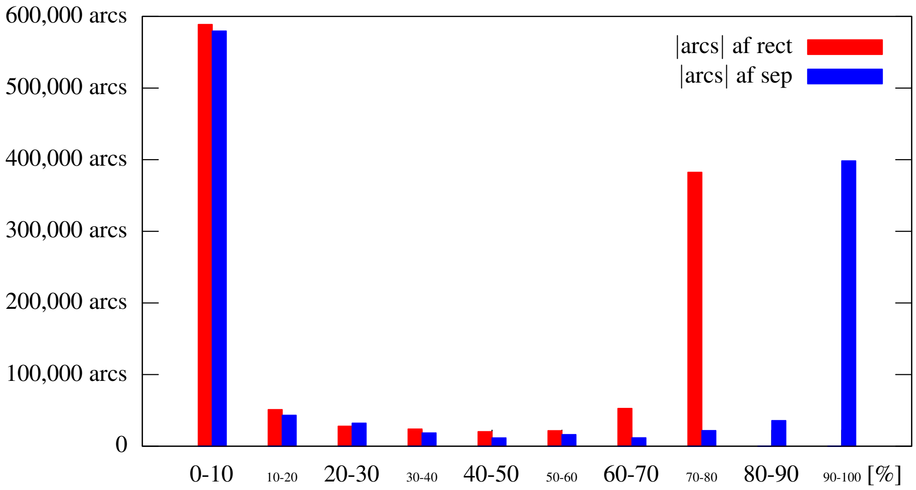

Fig. 8. Statistics of the fill rate of the flag vectors on instance OS (474,431 nodes, 1,169,224 arcs). The y-axis shows the number of arcs for which the according flag vector has a certain fill rate, while the x-axis shows the different fill rates in percentage. For instance, an arc a has a flag vector with fill rate 30% if 3 out of 10 flags in the vector are set to TRUE. It is remarkable that most of the arcs have either almost empty or almost full flag vectors.