# Partitioning Graphs to Speedup Dijkstra's Algorithm

ROLF H. MÖHRING and HEIKO SCHILLING
Technische Universität Berlin
and
BIRK SCHÜTZ, DOROTHEA WAGNER, and THOMAS WILLHALM
Universität Karlsruhe (TH)

We study an acceleration method for point-to-point shortest-path computations in large and sparse directed graphs with given nonnegative arc weights. The acceleration method is called the *arc-flag approach* and is based on Dijkstra's algorithm. In the arc-flag approach, we allow a preprocessing of the network data to generate additional information, which is then used to speedup shortest-path queries. In the preprocessing phase, the graph is divided into regions and information is gathered on whether an arc is on a shortest path into a given region. The arc-flag method combined with an appropriate partitioning and a bidirected search achieves an average speedup factor of more than 500 compared to the standard algorithm of Dijkstra on large networks (1 million nodes, 2.5 million arcs). 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. We conduct an experimental study that evaluates which partitionings are best suited for the arc-flag method. In particular, we examine partitioning algorithms from computational geometry and a multiway arc separator partitioning. The evaluation was done on German road networks. The impact of different partitions on the speedup of the shortest path algorithm are compared. Furthermore, we present an extension of the speedup technique to multiple levels of partitions. With this multilevel variant, the same speedup factors can be achieved with smaller space requirements. It can, therefore, be seen as a compression of the precomputed data that preserves the correctness of the computed shortest paths.

Categories and Subject Descriptors: G.2.2 [**Graph Theory**]: Graph algorithms; G.2.2 [**Graph Theory**]: Network problems; F.2.2 [**Nonnumerical Algorithms and Problems**]: Routing and layout

---

## 1. INTRODUCTION

In the present paper, we consider the *point-to-point shortest-path problem* where one has to find a shortest path between two nodes in a graph. The standard algorithm for this problem is the one developed by Dijkstra [1959], which runs in $\mathcal{O}(m + n \log n)$ time (see Fredman and Tarjan [1987]). For a long time, the main focus in developing shortest-path algorithms has been on finding algorithms with good theoretical time bounds. An overview is given in Goldberg and Harrelson [2005], Sanders and Schultes [2005], and Willhalm [2005]. Although fast in theory, the corresponding algorithms are often not fast enough for applications in large networks that require many shortest-path computations.

In our study, we assume that for the same underlying network the shortest-path problem has to be solved repeatedly for different node pairs. Thus, preprocessing of the network data is possible and can support the computations that follow. We work on large, but sparse, directed graphs with given arc weights and a given two-dimensional (2D) layout. We will see that the presented acceleration of shortest-path computations works also on higher dimensional layouts or even on graphs with no layout.

More precisely, we consider a generalization of a partition-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 [1997, 2004]. 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. The vector contains a flag for each region of the graph, indicating whether this arc is on a shortest path into that particular region. The vector is called the *arc-flag vector* and the entries in the arc-flag vector are called the *arc-flags*. 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.

### 1.1 Outline of the Paper

In Section 1.1, we give a brief review of recent related results in the field and in Section 1.2 we describe our contribution. Section 2 starts with basic definitions and a precise description of the problem. Section 3 explains the pruning of the search space of Dijkstra's algorithm with arc-flags. The preprocessing is described in Section 4. We discuss the two-level variant of the arc-flags in Section 5. In Section 6, we present the selection of partitioning algorithms that we used for our analysis. Section 7 describes our experiments and we discuss the results of the experiments in Section 8. Section 9 concludes the paper.

## 1.2 Related work

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. Therefore we can only give a short overview of some more recent results. Extensive surveys can be found in Goldberg and Harrelson [2005], Sanders and Schultes [2005], and Willhalm [2005].

Gutman [2004] 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. The reach value is then used to focus on nodes, which are part of a path long enough to be of use for the current shortest-path query. All other nodes are not taken into consideration during the search. Gutman reports that he computes shortest paths ten times faster than the standard algorithm of Dijkstra on networks containing about 400,000 nodes. Combined with an $A^*$ search, this method achieves a speedup factor of 17. On networks of the same size, our method runs about one order of magnitude faster and needs approximately the same preprocessing time. (See Figure 1c for a sample search in the German road network and the pruning of Dijkstra's search space generated by Gutman's acceleration technique.)

Goldberg and Harrelson [2005] describe an approach that uses an $A^*$ search in combination with a lower-bounding technique based on so-called *landmarks* and the triangle inequality. After selecting a small number of landmarks for all nodes, the distances to and from each landmark are precomputed. For instance, two one-to-all shortest-path computations per landmark suffice as preprocessing. The maximum of these lower bounds is used during an $A^*$ search. The speedup increases with the number of landmarks used for the search. With just one landmark, the results that Goldberg and Harrelson's algorithm produces are not as good as Gutman's, but with 16 landmarks they report on a speedup of up to 17 in networks up to a size of approx. 7 million nodes. The preprocessing for this method is very fast and the achieved speedups are reasonable, but the space requirement is large. One distance value per node-landmark pair needs to be saved.

Sanders and Schultes [2005] fairly recently published an interesting hierarchical approach. It is based on the idea of a *highway network*, which is defined by a local search. The local search visits a fixed number of nodes, which are closest to the terminal nodes. This approach can be iterated to generate a *hierarchy of highway networks*. The construction of the hierarchy can be done very rapidly in a preprocessing step. As in our approach, the space requirement is only a small constant factor of the input size. In terms of preprocessing time, this method is clearly faster than the arc-flag approach, but we still achieve speedups that are an order of magnitude higher on networks of the same size. 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. More examples can be found in Frederikson [1987], Agrawal and Jagadish [1994], Car and Frank [1994], Chou et al. [1998], Schulz et al. [2000], and Schulz [2005].

Schulz et al. [2000] already introduced the concept of enriching the graph with arc labels that mark (for each arc) possible target nodes of a shortest

(a) The standard algorithm of Dijkstra
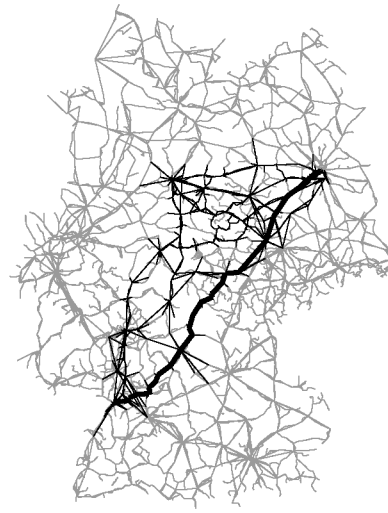
(b) The geometric container method

(c) The reach-based method
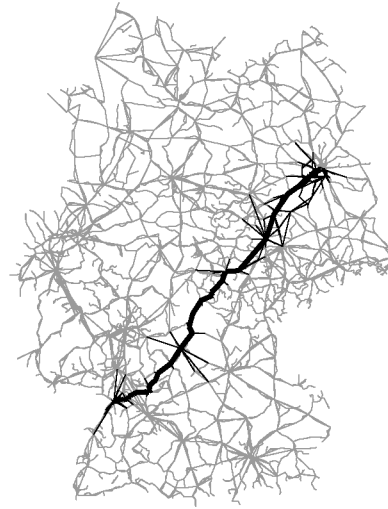
(d) The arc-flag method

Fig. 1. Different search spaces on the German road network for different acceleration methods compared to the standard algorithm of Dijkstra 1a. A sample search request from Karlsruhe to Berlin is shown. The network arcs and nodes are colored in light gray. The shortest-path arcs are colored in black. The search space, i.e., the arcs, which were visited by the shortest-path algorithms, is colored in dark gray.

path that start with this arc. This was done for the special case of a timetable information system. In this work, arc labels are angular sectors in the given layout of the train network. In Wagner and Willhalm [2003], and Wagner et al. [2005], the approach has been studied for general weighted graphs. Instead of the angular sectors, different types of convex geometric objects are implemented and compared. (See Figure 1b for a sample search in the German road

network and the pruning of Dijkstra's search space using arc labels and geometric containers.)

In theory, all mentioned acceleration techniques can be combined, but not all combinations would lead to an improvement of the speedup factor. Holzer et al. [2004] provide an extensive study of possible combinations of an $A^*$ search, bidirected search, the multilevel method, and geometric containers. They suggest different combinations for different graph types. In Schulz et al. [2000], for instance, a combination of a geometric container, the separator based multilevel method, and an $A^*$ search yields a speedup factor of 62 on a railway network.

A different variation of the arc labels has been presented by Lauther [2004], where the graph is first partitioned into regions by a simple rectangular geographic partition. Each arc then gets a vector of flags assigned where each entry corresponds to a region in the partition. An entry of the flag-vector is set to `true` iff the corresponding region contains target nodes of shortest paths starting with this arc. In contrast to the geometric containers [Wagner and Willhalm 2003; Wagner et al. 2005], arc-flags result in a much smaller search space and the generation of arc-flags can be realized without the computation of all-pairs shortest paths. In fact, only the distances to nodes are needed that lie on the boundary of a region (see Köhler et al. [2005] for more details).

Köhler et al. [2005] have shown that using an arc multiway separator partitioning instead of a simple rectangular geographic partitioning results in an even faster preprocessing of the arc-flags. At the same time, this improved partitioning led to better speedup factors of the accelerated shortest-path search itself. For instance, when a bidirected search is combined with the arc-flag method, Lauther [2004] obtains a speedup factor of 64 on the European truck drivers' road map (330,000 nodes, 520,000 arcs, 139 regions). Köhler et al., on the other hand, achieve a speedup factor of 677 on an instance of roughly the same size (360,000 nodes, 920,000 arcs, 100 regions). This is a result of the improved partitioning. Another advantage of the improved partitioning is that it does not require a given layout of the graph. In addition, Köhler et al. managed to use even fewer regions than Lauther and, therefore, reduced the space requirement of the arc-flag method.

An observation, which could not be expected at the beginning of the study conducted by Köhler et al. [2005], is that the choice of the underlying partition is crucial for the speedup of the arc-flag method. This is where the present paper begins.

## 1.3 Our Contributions

In this paper, we conduct a computational study on which partitioning method achieves the best speedup for the presented arc-flags. The choice of the underlying partitioning seems to be crucial for the speedup of the arc-flag acceleration of Dijkstra's algorithm. We investigated partitions from computational geometry and an arc multiway separator partitioning. The computational study was done on large road networks, a typical application for shortest-path problems. In particular, we worked on road networks of Germany up to the size of 1 million nodes and 2.5 million arcs. (See Figure 1d for a sample search in the German

road network and the pruning of Dijkstra's search space generated by the arc-flag approach, together with an arc multiway separator partitioning.)

Typically, the arc-flag accelerated algorithm of Dijkstra creates a conelike spreading of the search space as the search approaches the target region; whereas at the beginning of a shortest-path search, the algorithm is forced by the arc-flags to search along the shortest-path arcs only. In order to handle this behavior of the arc-flag method, we suggest a two-level partition: a coarse partition for far-away target nodes and a finer one for nearby nodes. With this multilevel version of the arcflags, the same speedup can be achieved, but with lower space consumption. The multilevel arc-flags can be seen as a compression of the flagvectors.

Finally, we tested different combinations of the the arc-flag method with different partitions and other acceleration techniques. We found that a combination with a bidirected search seems to be a perfect match, but a combination with an $A^*$ search, for example, does not improve the speedup factor. The reason is that the arc-flags are already highly goal-directed by construction.

## 2. DEFINITIONS AND PROBLEM DESCRIPTION

### 2.1 Graphs

A directed simple *graph G* is a pair $(V, A)$, where $V$ is a finite set of *nodes* and $A \subseteq V \times V$ are the *arcs* of the graph $G$. Throughout this paper, the number of nodes $|V|$ is denoted by $n$ and the number of arcs $|A|$ is denoted by $m$. A *path* in $G$ is a sequence of nodes $u_1, \ldots, u_k$ such that $(u_i, u_{i+1}) \in A$ for all $1 \leq i < k$. A path with $u_1 = u_k$ is called a *cycle*. A graph (without multiple arcs) can have up to $n^2$ arcs. We call a graph *sparse*, if $m \in \mathcal{O}(n)$. We assume that we are given a *layout* $L : V \to \mathbb{R}^2$ of the graph in the Euclidean plane. For ease of notation, we will identify a node $v \in V$ with its location $L(v) \in \mathbb{R}^2$ in the plane.

### 2.2 The Shortest-Path Problem

Let $G = (V, A)$ be a directed graph whose arcs are *weighted* by a function $\ell : A \to \mathbb{R}$. We interpret the weights as *arc lengths* in the sense that the *length* of a path is the sum of the weights of its arcs. The *(single-source single-target) shortest-path problem* consists in finding a path of minimum length from a given source $s \in V$ to a given target $t \in V$. Note that the problem is only well defined for all pairs, if $G$ does not contain negative cycles. If there are negative weights, but not negative cycles, it is possible, using Johnson's algorithm [Johnson 1977], to convert in $\mathcal{O}(nm + n^2 \log n)$ time the original arc weights $\ell : A \to \mathbb{R}$ to nonnegative arc weights $\ell' : A \to \mathbb{R}_0^+$ that result in the same shortest paths. Hence, in the rest of the paper, we can safely assume that arc weights are nonnegative. Throughout the paper we also assume that for all pairs $(s, t) \in V \times V$ the shortest path from $s$ to $t$ is unique. Kranakis et al. [1995], for instance, gave a detailed description on how one can transform any graph into such a unique shortest-path graph.
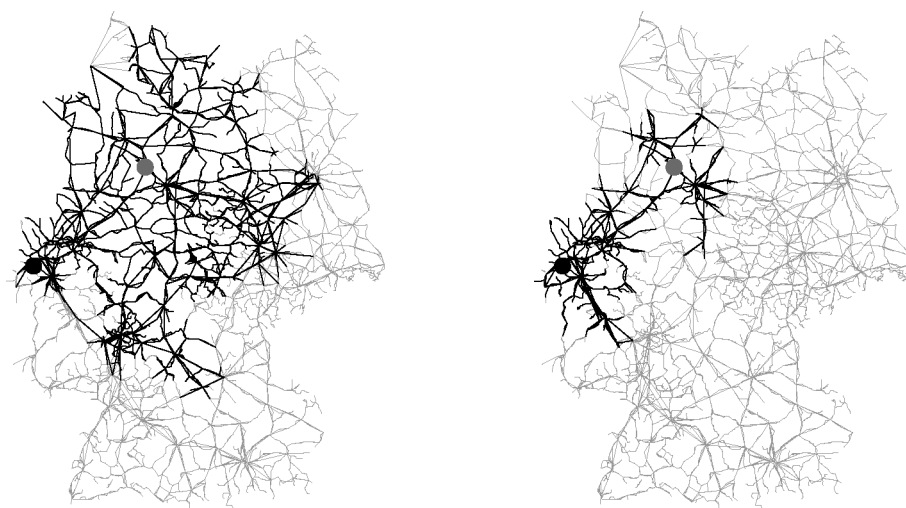
Fig. 2. This figure illustrates the search space of a unidirectional standard Dijkstra search (left) and a bidirectional standard Dijkstra search (right). The source node is marked in light gray, the target node in dark gray, and all arcs which are touched during the search are the dark gray arcs. The search stops when the target node has been settled.

## 2.3 Bidirectional Search

In the *bidirected search* two Dijkstra runs start simultaneously from $s$ and $t$. A distance $dist_s(u)$ from $s$ in the common (forward) graph and a distance $dist_t(u)$ from $t$ in the *reverse graph*, the graph with every arc reversed, is then computed. The bidirected search algorithm alternates between running the common (forward) 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.

More precisely, the bidirectional search stops if one direction gets a node $v$ from the priority queue that is already labeled by the other direction: the shortest path between $s$ and $t$ is already found. The node $v$ is not necessarily on that shortest path. In order to avoid searching for the connector node $v$ of the two searches, we determine the shortest path on-the-fly: every time we consider a node, which is labeled by both directions, we update the minimal sum of the shortest paths to source and target.

The bidirectional search leads to speedup factors of up to 4 in the unaccelerated case. Figure 2 illustrates the search space in contrast to the unidirectional Dijkstra search. In principle, this speedup method can be combined with any other acceleration method. In our experiments, a forward and backward accelerated bidirectional search achieved the best results. This means that we applied the partition-based speedup technique on both search directions with one-half of the arc-flag entries for each direction. The underlying partitioning can differ for the two directions. The preprocessing for both directions must be computed independently. The reverse graph is used for the calculation of the arc-flags for the backward search.
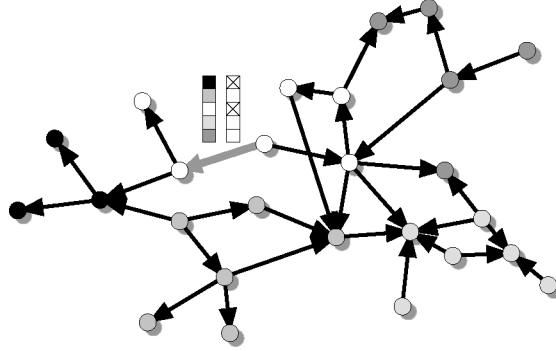
Fig. 3.   Illustration of the partition-based arc-flags acceleration: the labeled arc (i.e., the gray arc in the figure) only leads to white and black nodes. The according arc-flag vector of this arc has, therefore, only two entries: one for the region of the black nodes and one for the region of the white nodes. A search with targets in the light, middle, and dark gray regions can ignore this arc.

## 3. DIJKSTRA'S ALGORITHM WITH ARC-FLAGS

The classical algorithm for computing shortest paths in a directed graph with nonnegative arc weights is that developed by Dijkstra [1959]. For the general case of arbitrary nonnegative arc lengths, it still seems to be the fastest algorithm with $\mathcal{O}(m + n \log n)$ worst-case time. However, in practice, speedup techniques can reduce the running time and often result in a sublinear running time. They crucially depend on the fact that Dijkstra's algorithm is label setting and that it can be terminated when the destination node is settled. Therefore, the algorithm does not necessarily search the whole graph.

If we allow for a preprocessing step, the running time can be further reduced with the following insight: consider, for each arc $a$, the set of nodes $S(a)$ that can be reached by a shortest path starting with $a$. It is easy to verify that Dijkstra's algorithm can be restricted to the subgraph with those arcs $(*, t) \in E$ for which the head node $t$ is in $S(a)$. However, storing all sets $S(a)$ requires $\mathcal{O}(n^2)$ space, which is prohibitive for large graphs even if they are sparse ones with $m \in \mathcal{O}(n)$ (as in our case). We will, therefore, use a partition of the set of nodes $V$ into $p$ regions for an approximation of the set $S(a)$. Formally, we will use a function $r : V \rightarrow \{1, \ldots, p\}$ that assigns to each node the number of its region. For example: given a 2D layout of the graph, a simple method to partition a graph is to use a regular grid as illustrated in Figure 6a (see later) and assign the same number to all nodes inside a grid cell.

We will then use for each arc $a$, a vector $f_a : \{1, \ldots, p\} \rightarrow \{$ true, false$\}$ with $p$ entries, each of which corresponds to a region. The vector $f_a$ will be called the *arc-flag vector*. The entries in the vector will be called the *arc-flags*. For each arc $a$, we set the arc-flag $f_a(i)$ to true iff $a$ is the beginning of a shortest path to at least one node in region $i \in \{1, \ldots, p\}$. (See Figure 3 for an illustration of an instantiated arc-flag-vector.) For a specific shortest-path query from $s$ to $t$, Dijkstra's algorithm can be restricted to the subgraph $G_t$ with those arcs $a$ for which the flag entry of the target-region (i.e., the region where $t$ belongs to) is set to true. We will call this restricted version of Dijkstra's algorithm

the *Dijkstra's algorithm with arc-flags*. Lemma 3.2 provides the correctness of Dijkstra's algorithm with arc-flags and for this we need to define the notion of a *consistent container*.

*Definition* 3.1. Let $G = (V, E)$ be a weighted graph together with a weight function $\ell : E \to \mathbb{R}$. We call a set of nodes $C \subseteq V$ a *container*. A container $C$ associated with an arc $(u, v)$ is called *consistent*, if for all shortest paths from $u$ to $t$ that start with the arc $(u, v)$, the target node $t$ is in $C$.

LEMMA 3.2 (DIJKSTRA'S ALGORITHM WITH ARC-FLAGS). *Let $G = (V, E)$ be a weighted graph together with a weight function $\ell : E \to \mathbb{R}$. Then, for each arc, $e \in E$ the nodes in the regions, which are associated with the* `true` *entries of the arc-flag vector of e, constitute a consistent shortest-path container.*

PROOF. Consider the shortest path $P$ from $s$ to $t$ that is found by Dijkstra's algorithm. If for all arcs $e \in P$, the target node $t$ is in $C(e)$, then the path $P$ will also be found by Dijkstra's algorithm with arc-flags. This is because the arc-flags do not change the order in which the arcs are processed. A subpath of a shortest path is again a shortest path, so for all arcs $(u, v) \in P$, the subpath of $P$ from $u$ to $t$ is a shortest $u, t$-path. Therefore, by definition of the consistent container: $t \in C(u, v)$ and so Dijkstra's algorithm with arc-flags finds a shortest path from $s$ to $t$, $s, t \in V$, if one exists. □

The subgraph $G_t$ can be computed online during a run of Dijkstra's algorithm. In a shortest-path search from $s$ to $t$, while scanning a node $u$, the modified algorithm takes into account all the outgoing arcs that have their arc-flag entries set to `true` for the region of the target node $t$. All other arcs will be ignored. All possible partitions of the nodes lead to a correct solution, but most of them would not lead to the desired speedup of the computation.

The space requirement for the preprocessed data is $\mathcal{O}(pm)$ for $p$ regions because we have to store one flag for each region and arc. If $p = n$ and we assign to every node its own region number, we store, in fact, all-pairs shortest paths: if a node is assigned to its own, specific region, the modified shortest-path algorithm will find the direct path without looking at unnecessary arcs or nodes. Note, however, that, in practice, even for $p \ll n$, we achieve an average search space that is only four times the number of nodes in the shortest path. Furthermore, it is possible within the framework of the arc-flag speedup technique to use a specific region only for the most important nodes. Storing the shortest paths to important nodes can, therefore, be realized without any additional implementation effort. It is common practice to cache the shortest paths to the most important nodes in the graph.

## 4. PREPROCESSING

We have to calculate the arc-flag entries for all arcs. This can be achieved by computing a shortest path tree from every arc $a$ to all nodes in the graph—a one-to-all shortest-path computation from the head node of arc $a$. The computation is done by a standard algorithm of Dijkstra, which stops when all nodes are permanently marked. During this computation, if a node $v$ is settled, we

are setting the arc-flag entry $f_a(r(v))$ to `true` for the region $r(v)$ containing node $v$.

The setting of the bit-vectors needs $\mathcal{O}(mn)$ time for all pairs of arcs and nodes. The running time for the preprocessing is dominated by the time needed to compute $m$ times a shortest-path tree, which can be done in $\mathcal{O}(m + n \log n)$ time each. The resulting time complexity of the overall preprocessing is, therefore, $\mathcal{O}(m(m + n + n \log n))$. For sparse graphs ($m = \mathcal{O}(n)$), such as typical road networks, we get a worst-case time complexity of $\mathcal{O}(n^2 \log n)$. For large $n$, this simple version of the preprocessing algorithm takes far too long. Therefore, we are heading for a more scalable preprocessing algorithm, which takes subquadratic or even linearithmic running time. Fortunately, we can give a variant of the preprocessing with a much better (although not linearithmic) running time.

## 4.1 Preprocessing Without All-Pairs Shortest-Path Computations

It is not necessary to compute all-pairs shortest paths to fill the flag vectors correctly. We can simply use the following insight: every shortest path from any node $s$ to a region $R$ with the region number $p_R$ has to enter the region $R$ by an arc: if $s$ is not a member of the region $R$, then there must be an arc $e = (u, v)$ with $r(u) \neq p_R$ and $r(v) = p_R$: a so-called *overlapping arc*. We will see in Lemma 4.1 that in the preprocessing step it is sufficient (for the correct computing of the arc-flags) to take into account only shortest paths to such nodes $v$, which are head nodes of overlapping arcs. Such nodes will be called *boundary nodes*.

LEMMA 4.1 (BOUNDARY NODES). *Let $G = (V, E)$ be a graph and let $r : V \to \{1, \dots, p\}$ define a partition of the node set $V$ in $p$ regions. Let further for all arcs $(u, v) \in E$, where the end nodes $u$ and $v$ belong to the same region $r'$ the flag vector entry $f_{(u,v)}(r')$ set to `true`. If the arc-flag vectors $f_e$ of all arcs $e \in E$ are computed with the set of shortest paths to boundary nodes only, then the nodes in the regions, which are associated with the `true` entries of the arc-flag vectors, are consistent shortest-path containers.*

PROOF.    Let $s$ and $t$ be arbitrary, but fixed, nodes, which are connected by the shortest path $s = n_1, \dots, n_k = t$. Further, let $s$ and $t$ belong to different regions $p_s = r(s) \neq r(t) = p_t$. By induction, one can easily see that there exists an arc $e = (n_i, n_{i+1}), 1 \leq i < k$, in this shortest path with $p_{n_i} = r(n_i) \neq r(n_{i+1}) = p_t$. The preprocessing, which only considers shortest paths to boundary nodes, would have considered the path from $s$ to node $n_{i+1}$ and, hence, it would have set the flag-vector entry, which refers to region $p_t$ for all arcs of the shortest path $s, \dots, n_{i+1}$. The flag-vector entry, which refers to region $p_t$ of the arcs between $n_{i+1}$ and $t$, are set to `true` by definition, because these arcs belong to the region $p_t$. Since the flag entries for the target region $p_t$ are set for all arcs on the shortest path $s = n_1, \dots, n_k = t$, the modified algorithm of Dijkstra finds the shortest path from $s$ to $t$.    □

For the processing of the shortest paths to boundary nodes, we define the *reverse graph* $D_{\text{rev}}$ of a directed graph $D = D(V, E, l)$ as the graph $D_{\text{rev}} =$

$D(V, E_{\text{rev}}, \ell_{\text{rev}})$ with

$$E_{\text{rev}} = \{(u, v) | (v, u) \in E\} \quad \text{and} \quad \ell_{\text{rev}}(u, v) = \ell(v, u)$$

Hence, the reverse graph is the graph $D$ with all arcs reversed. It is easy to see that $s, \ldots, t$ is a shortest path from $s$ to $t$ in $D$ iff $t, \ldots, s$ is a shortest path in $D_{\text{rev}}$ with the same arcs reversed.

We can now exploit this property. The preprocessing algorithm determines all boundary nodes of the chosen partition and calculates their shortest path trees in the reverse graph $D_{\text{rev}}$. More precisely, let us look at one region $R$ and its boundary node set

$$B_R = \{v \in R | \exists (u, v) \in E \;\; \text{such that} \; r(u) \neq r(v)\}$$

For each node $b \in B_R$, the algorithm calculates shortest-path trees in $D_{\text{rev}}$. For each adjacent node of $b$, which does not belong to $B_R$, the preprocessing algorithm calculates one shortest-path tree in $D_{\text{rev}}$. It is necessary to do one tree computation per adjacent node of $b$ (outside $B_R$), because otherwise one could miss to set true entries in the flag vectors and this could later lead to wrong shortest-path computations. The preprocessing algorithm stores for each node $w$ in a shortest-path tree its incoming arc $e_w$ of the shortest-path tree. In $D_{\text{rev}}$ this is an incoming arc of $w$ and, hence, in $D$ it is an outgoing arc, which is the beginning of the shortest path from $w$ to $b$ in $D$. For this outgoing arc $e$, the algorithm sets the region flag of $R$ in the flag vector of $e$, because there exists a shortest path, which starts with $e$ and ends in region $R$. After scanning all nodes of $B_R$, all shortest paths ending in region $R$ have been taken into account and all region flags of region $R$ of all arcs in the graph have been set.

This is an improvement of the first preprocessing algorithm, because the present algorithm does not calculate the shortest-path trees of all nodes of $D$— only the shortest-path trees of overlapping arcs are calculated. We denote the number of overlapping arcs by $k$. Depending on $k$ we get a time complexity of the improved preprocessing of $\mathcal{O}(kn \log n)$. An example of the resulting preprocessing time is the following: the improved preprocessing algorithm calculated the preprocessing data of one of our real-life graphs with 473,000 nodes and 1,1 million arcs by using a $10 \times 10$ grid partition in 2.5 hours. The average search space during a single requests was reduced to less than 4% compared to the standard Dijkstra search—this led to a factor 34 improvement of shortest-path computation time. The number $k$ of overlapping arcs is highly dependent on the partitioning of the nodes. We can minimize $k$ by taking the minimum arc multiway separator for a partitioning of the graph. However, we will see in Section 8 that this partition does not always lead to the best results with regard to the size of the search space, when we perform an accelerated shortest path search.

For a further reduction of the preprocessing time, one can easily adapt the preprocessing algorithm to a parallel algorithm: each processor calculates the shortest-path trees of a subset of the boundary nodes $B_R$. The results are independent.

Table I. Analysis of the arc-flags.[a]

| Graph | #Arcs | Algorithm | = 1 | < 10% | > 95% |
|---|---|---|---|---|---|
| road_network_1 | 920,000 | KdTree(32) | 351,255 | 443,600 | 312,021 |
| road_network_1 | 920,000 | KdTree(64) | 334,533 | 470,818 | 294,664 |
| road_network_1 | 920,000 | METIS(80) | 346,935 | 468,101 | 290,332 |
| road_network_4 | 2,534,000 | KdTree(32) | 960,779 | 1,171,877 | 854,670 |
| road_network_4 | 2,534,000 | KdTree(64) | 913,605 | 1,209,353 | 799,206 |

[a] kdTree($n$) and METIS($n$) are partitioning algorithms of size $n$ (see also Section 8). For 80% of the arcs, either almost none ($< 10\%$) or nearly all ($> 95\%$) flags of the corresponding flag vector have been set to true.

## 4.2 Preprocessing with Pruned Shortest-Path Trees

In this section, we present a technique to avoid the computation of the full shortest-path trees of all boundary nodes. Let $v_1, v_2 \in V$ be two boundary nodes of the same region. When we compute the shortest path tree for $v_1$, we get an upper bound on the length of the shortest path from $v_2$ to $v$ for every node $v \in V$: it cannot be longer than the distance from $v_2$ to $v_1$ plus the shortest path length from $v_1$ to $v$. We can now use this upper bound to reduce the effort of preprocessing.

For the first boundary node $v_1$ of each region, the algorithm computes its (reverse) shortest path tree, the corresponding flag vectors, and distance from the closest boundary node $v_2$ of that region. During the computation of the shortest path tree of $v_2$, we use the upper bounds of the prior search: if we find a shortest path to a node $v$, which is longer than or equal to the upper bound, the algorithm does not put $v$ into the priority queue, because we will get no new information about our flag vectors—we have already found the shortest path to that node $v$. If a shorter path is found later, the node will be put into the priority queue and the algorithm provides correct results. During the computation of the shortest-path tree for the next boundary node $v_3$, upper bounds are computed using $v_2$, and so on.

Our experiments have shown that this method reduces the number of nodes that are put into the priority queue during the preprocessing to less than 70%, compared to the preprocessing from the previous section. In the experiments, the running time of the preprocessing was improved by up to 20%.

## 5. TWO-LEVEL PARTITIONS

An analysis of the calculated flag vectors shows that it is possible to find a (lossy) compression of the flag vectors: for 80% of the arcs either almost none or nearly all flags of their flag vectors are set to true. Table I and Figure 4 show an example of the analysis we made.

The column "= 1" shows the number of arcs, which are only part of shortest paths inside their own region (only one flag has been set to true). Arcs with more than 95% of arc-flag entries set to true seem to be central in the graph and could be important roads or could be incident to important intersections in the road
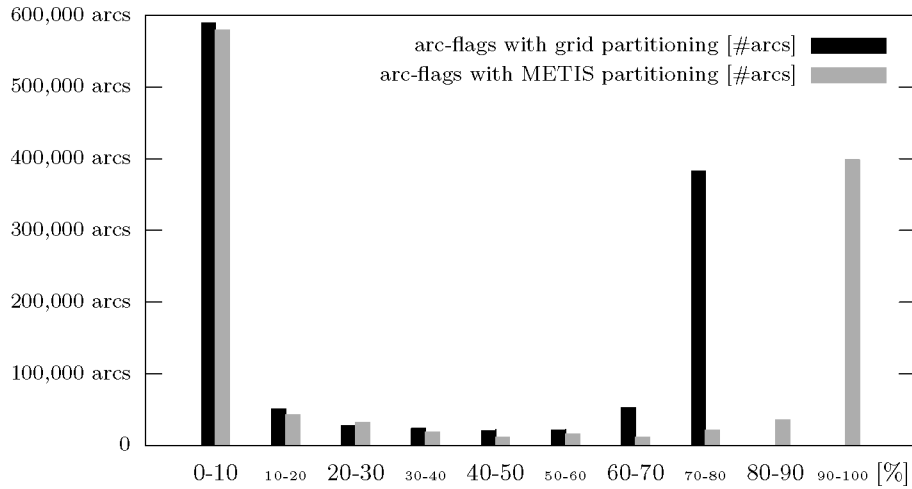
Fig. 4. Statistics of the fill rate of the flag vectors on instance road_network_2 (1,169,000 arcs). The y axis shows the number of arcs for which the corresponding flag vector has a certain fill rate, while the x axis shows the different fill rates in percentage. For instance, an arc $a$ has a flag vector with fill rate 30% if 3 out of 10 flags in the vector have been set to true.

network. Obviously, these results are highly dependent on the characteristics of the graphs, but the situation will probably be similar for other road networks, too.

The high amount of almost empty flag-vectors justifies the idea for a compression of the vectors. It is important that the decompression algorithm is very fast—otherwise the speedup of the running time will be lost. The two-level technique, described in the following section, is a suitable lossy compression method for the flag-vector entries.

Let us have a closer look at a search space to get an idea of how to compress the arc-flags. As illustrated in Figure 5a, for a search from the dark gray node to the light gray node, the accelerated Dijkstra search reduces the search space at the beginning of the search, but once the target region has been reached, almost all nodes and arcs are visited. This is not very surprising if we consider that usually all arcs of a region were assigned the region flag of their own region. We could deal with this problem by using a finer partition of the graph, but this would lead to longer flag vectors at each arc (requiring more memory and a longer preprocessing). Take the following example, if we used a finer $15 \times 15$ grid instead of a coarse $5 \times 5$ grid, each coarse region would then be split in nine additional finer regions. The preprocessing data, however, increases from 25 flags (in the coarse case) to 225 flags (in the finer case) per arc. We can see that the additional information for the fine grid is mainly needed for arcs close to the target node, e.g., arcs in the target region of the coarse grid. This leads to the idea that we could split each region of the coarse partition, but store the additional data (for the fine grid) only for the arcs inside the same coarse region. More precisely, we regard the target region as an independent graph. We can partition this independent graph again (see Figure 5b), and perform an additional preprocessing for the flag vectors at each arc in this graph.

(a) With one-level arc-flags a search visits almost all arcs in the region where the target node belongs to. The target node is dark gray. The source node is light gray. Visited arcs are dark gray.

(b) With the two-level arc-flags for each arc $a \in E$ a flag vector is stored for a coarse partition ($5 \times 5$ grid) and another flag vector for a fine partition ($3 \times 3$ grid) *of the coarse region where the arc $a$ belongs to.*
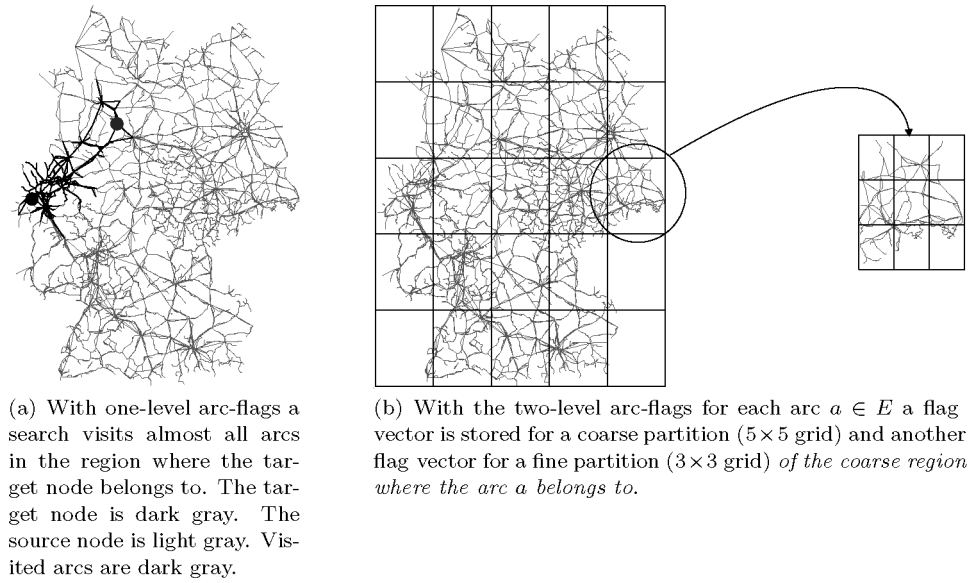
Fig. 5. The one- versus two-level arc-flag method.

Therefore, each arc gets two flag vectors: one for the coarse partition and one for the associated regions of the fine partition. (See the pseudocode of the *modified algorithm of Dijkstra with arc-flags and two-level partition* for more details.) In the pseudocode, the vectors *ArcFlagVectorFirstLevel* and *ArcFlagVectorSecondLevel* refer to the flag vectors for the coarse and the fine partition. Entries in these vectors are set to true) for an arc $(v, u) \in E$ if the arc $(v, u)$ is at the beginning of a shortest path into the *TargetRegion* (for the coarse partition and ArcFlagVectorFirstLevel) or into the *SubTargetRegion* (for the fine partition and ArcFlagVectorSecondLevel).

The advantage of this method is that the preprocessed data is smaller than for a fine one-level partitioning, because the second flag vector exists only for the target region (34 flags per arc instead of 225). It is clear that the $15 \times 15$ grid would lead to better results. However, the difference between the search spaces is small since the entries in flag vectors of far-away neighboring regions are similar. Thus, one can see this two-level method as a (lossy) compression of the first-level flag vectors: we summarize the flags for remote regions. If one flag is set for a fine region, the flag is set for the whole group.

Only a slight modification of the search algorithm is required. Until the target region is reached, everything will remain unaffected, unnecessary arcs will be ignored with the flag vectors of level one. When the algorithm has entered the target region, the second-level flag vector provides further information on whether an arc can be ignored for the search of a shortest path to the target node.

As we will see in Section 8, our experiments have shown that this method led to the best results with regard to the reduction of the search space, but an increased preprocessing effort is required. Note, however, that in the

---

**Modified algorithm of Dijkstra with arc-flags and two-level partition**

---

**Input**: directed graph $G = (V, A)$, nonnegative length $\ell_a$ for all $a \in A$,
        start and target nodes $s, t \in V$
**Output**: shortest path from $s$ to $t$
**begin**
    TargetRegion := region number of t; //coarse partition
    SubTargetRegion := subregion number of t; //fine partition
    distance$(s) := 0$;
    Queue.insert$(s,0)$;
    **while not** Queue.empty **do**
        $v :=$ Queue.extractMin;
        **for** all outgoing arcs $(v, u)$ **do**
            **if not** ArcFlagVectorFirstLevel$[(v, u)$,TargetRegion$]$ **then**
                **continue**;

            **if** $(v, u) \in$ TargetRegion **then**
                **if not** ArcFlagVectorSecondLevel$[(v, u)$,SubTargetRegion$]$
                **then**
                    **continue**;

            **if** $distance(u) \leq distance(v) + \ell_{(v,u)}$ **then**
                **continue**;

            $distance(u) = distance(v) + \ell_{(v,u)}$;
            **if** $u \notin Queue$ **then**
                Queue.insert$(u)$;

            **else**
                Queue.decreaseKey$(u)$;

**end**

---

preprocessing it is not necessary to compute the complete shortest path trees for all boundary nodes of the fine partitioning. The computation can be stopped if all nodes in the same coarse region have been permanently marked.

## 6. PARTITIONING ALGORITHMS

The arc-flag speedup technique uses a partitioning of the graph to precompute information on whether an arc may be part of a shortest path. Any possible partitioning can be used for the technique and the accelerated Dijkstra algorithm will always return a correct shortest path. However, different partitions do lead to different accelerations of the algorithm of Dijkstra. The question is which partitions lead to the best speedup.

   In this section, we will present the partitioning algorithms that we used in combination with the arc-flag approach. Most of these algorithms need a 2D layout of the graph except for the multiway arc separator algorithm used by METIS. The partitioning algorithms based on a 2D layout can easily be adapted to higher dimensions. In fact, for the arc-flag approach itself no layout of the graph is necessary as long as one can provide a partitioning for a given graph.

### 6.1 Rectangular Partitioning (Grid)

Probably the easiest way to partition a graph with a 2D layout is to define the regions with a $n \times m$ grid of the bounding box. More precisely, we denote with $(\ell, t)$ the top-left coordinate of the bounding box of the 2D layout of the graph and with $(r, b)$ the bottom-right one. Furthermore, we define $w = r - \ell$ as the width and $h = t - b$ as the height of the layout. The grid cell or region $G_{i,j}$ with $0 \le i < n, 0 \le j < m$ is now defined as the rectangle

$$\left[ \ell + i\frac{w}{n} ; \ell + (i+1)\frac{w}{n} \right] \quad \times \quad \left[ b + j\frac{h}{m} ; b + (j+1)\frac{h}{m} \right]$$

Nodes on a grid line are assigned to one of the neighboring grid cells. Figure 6a shows an example of a $7 \times 5$ grid.

Arc-flags for a grid can be seen as a *raster image* of $S(u, v)$, where $S(u, v)$ represents the set of nodes $x$ for which the shortest $u$-$x$ path starts with the arc $(u, v)$. The pixel $i$ in the image is set, iff $(u, v)$ is the beginning of a shortest path to a node in region $i \in \{1, \ldots, p\}$. A finer grid, i.e., an image with higher resolution, provides a better image of $S(u, v)$, but requires more memory. On the other hand, the geometric objects by Wagner and Willhalm [2003, 2005] approximate $S(u, v)$ by a single convex object of constant size.

The rectangular or grid partitioning method only uses the bounding box of the graph—all other properties like the structure of the graph or the density of nodes are ignored and, hence, it is not surprising that this method does not lead to the best partitioning for our application. In fact, the rectangular partitioning always has the worst results in our experiments. Since earlier work on the arc-flag method [Lauther 2004] used the rectangular partitioning, we use it as a baseline and compare all other partitioning algorithms with it.

### 6.2 Quad Trees

A *quad tree* is a data structure for storing points in the plane. Quad trees are typically used in algorithmic geometry for range queries, since they support fast access to nearest neighbor points. Further applications are in computer graphics, image analysis, and geographic information systems. Quad trees can be generalized to higher dimensions—for 3D they are called *oct trees*.

*Definition* 6.1 (*QuadTree*).    Let $P$ be a set of $n$ points in the plain and $R_0$ its quadratic bounding-box. Then the data structure *quad tree* is recursively defined as follows:

- Root $v_0$ corresponds to the bounding-region $R_0$
- $R_0$ and all other regions $R_i$ are recursively divided into four quadrants, while they contain more than one point of $P$. The four quadratic subregions of $R_i$ are subnodes of $v_i$ in the tree. (See Figure 7)

The leaves of a quad tree form a subdivision of the bounding-box $R_0$. Even more, the leaves of every subtree containing the root form such a subdivision. Since, for our application, we do not want to create a separate region for each node, we use a subtree of the quad tree. More precisely, we define an upper

(a) Rectangular partition (35 regions)

(b) Quad-tree (34 regions)

(c) $kd$-Tree (32 regions)

(d) METIS (32 regions) Different colors correspond to different regions.
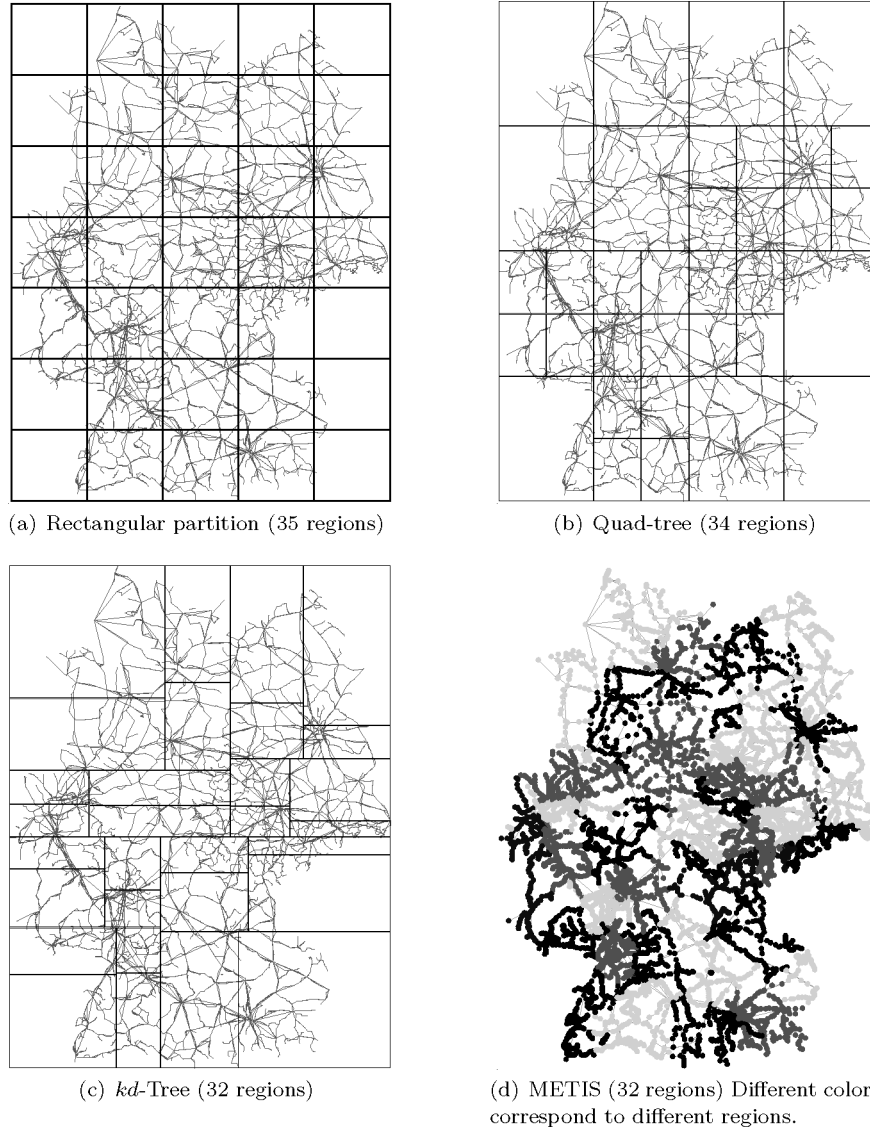
Fig. 6.   Germany with four different partitions.

bound $b \in \mathbb{N}$ of points in a region and stop the division if a region contains fewer points than the bound $b$. The result is a partition of our graph, where each region contains, at most, $b$ nodes. Figure 6b shows such a partition with 32 regions. In contrast to the grid partition, this partitioning reflects the geometry of the graph better—dense parts will be divided into more regions than sparse parts.

### 6.3 $kd$ Trees

In the construction of a quad tree, a region is divided into four equally sized subregions. However, equally sized subregions do not take the distribution of
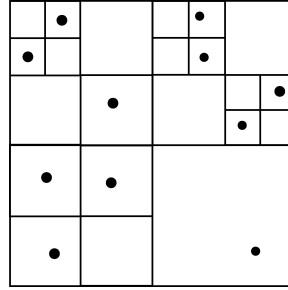
Fig. 7. Example of a quad tree. Each region is recursively divided until each region contains only one point.

the points into account. This quad-tree division can, therefore, be extended to more general subdivision schemes, the so-called *kd trees*. In the construction of a *kd tree*, the plane is recursively divided in a similar way as for a quad tree. In contrast to a quad tree, the underlying rectangle is decomposed into *two halves* by a straight line parallel to an axis. The axes alternate in the order $x, y, x, y, \ldots$. The positions of the dividing line can depend on the data. Frequently used positions are given by the center of the rectangle (*standard kd* tree), the *average*, or the *median* of the points inside. If the median of points in general position is used, the partitioning has always $2^\ell$ regions. Figure 6c shows a result for the median and 32 regions. In applications with higher dimensions, the partition axes are not cycled, but the dimension with the largest variance is used.

Our experiments on real-life road networks showed that the *kd tree* with median position partition usually leads to the best results (see Figure 6c). Therefore, we only used this method as a representative for this partitioning class. If the median of the points is used, at every decomposition, one node of the graph lies exactly on the boundary of two regions. For these nodes, it is worthwhile to check whether all neighbors of that node have their positions in the other region. If yes, the node can be transferred to the other region and will not become a boundary node.

The median of the nodes can be computed in linear time with the *median of medians* algorithm [Cormen et al. 2001]. Since the running time of the preprocessing is dominated by the shortest path computations after the partitioning of the graph, we decided to use standard algorithms: sorting the nodes and taking the mean. As an example, the *kd-tree* partitioning with 64 regions for one of our test graphs with one million nodes was calculated in 175 s; the arc-flags were calculated in 7 hours.

## 6.4 METIS

A fast method to partition a graph into $R$ almost equally sized sets with a small cut set is presented by Karypis and Kumar [1998]. An efficient implementation can be obtained free of charge from Karypis [1995]. There are two advantages of this method for our application. The METIS partitioning does not need a layout of the graph and the preprocessing is faster because the number of arcs in the

Table II. Overview of the Tested Algorithms

| Name | Description | Parameter |
|------|-------------|-----------|
| Grid | $c \times c$ grid over graph layout | $c$ |
| KdTree | $kd$ tree concerning coordinates of nodes | depth of $kd$ tree |
| METIS | partition generated by METIS | number of regions |
| 2LevelGrid | $c \times c$ grid coarse grid, $g \times g$ fine grid | $c$ and $g$ |
| 2LevelKdTree | coarse and fine $kd$ tree | depth of coarse and fine $kd$ tree |
| 2LevelMETIS | coarse METIS and fine METIS | number of coarse and fine regions |
| BiGrid | bidirectional grid | size of forward and backward grid |
| BiKdTree | bidirectional $kd$ tree | depth of $kd$ trees |
| Bi2LevelGrid | bidirectional 2LevelGrid | sizes of grids |
| Bi2LevelKdTree | bidirectional 2LevelKdTree | depth of $kd$ trees |
| BiMETIS | bidirectional METIS | number of forward and backward regions |

cut is noticeably smaller than in the other partitioning methods. Figure 6d shows a partitioning of a graph generated by METIS.

## 7. EXPERIMENTAL SETUP AND IMPLEMENTATION

The main goal of the present work is to compare four different partitioning algorithms in combination with the arc-flag acceleration method for Dijkstra's algorithm. The comparison is done with regard to the resulting search space and speedup of processing time of the accelerated Dijkstra search. We have four orthogonal dimensions in our algorithm tool box:

1. The base partitioning method: Grid, $kd$ Tree, or METIS
2. The number of partitions
3. Usage of one-level or two-level partitions
4. A unidirectional or a bidirectional search

As the differences between quad trees and $kd$ trees are very small, we used only $kd$ trees with median in the rest of this work, as a representative for this partitioning class.

We chose eleven combinations of the above-described techniques, which we believe are the most promising ones. For instance, we left out a combination of two-level arc-flags with a METIS partitioning and a bidirectional search (Bi2Metis), because, in most cases, the two-level arc-flags in a bidirectional search hardly performed better than the one-level variant. Table II shows our selection of combinations.

Furthermore, we fixed the size of the preprocessed data to nearly the same number for all algorithms. Exactly the same size cannot be realized because of the restrictions by the construction of the partitioning algorithms. Table III shows the preprocessed data size for the algorithms we tested.

We looked at the algorithms on German road networks, which are directed and have a 2D layout and positive arc weights. Table IV shows some characteristics of the graphs. The column "shortest path" refers to the average number of

Table III.  Partitions with Nearly the Same Preprocessed Data Size of 80 Bit

| Name of Partitioning | Forward 1st Level | Forward 2nd Level | Backward 1st Level | Backward 2nd Level | Bits Per Arc |
|---|---|---|---|---|---|
| Grid | $9 \times 9$ | — | — | — | 81 |
| KdTree | 64 | — | — | — | 64 |
| METIS | 80 | — | — | — | 80 |
| 2LevelGrid | $8 \times 8$ | $4 \times 4$ | — | — | 80 |
| 2LevelKd | 64 | 16 | — | — | 80 |
| 2LevelMETIS | 72 | 8 | — | — | 80 |
| BiGrid | $7 \times 7$ | — | $6 \times 6$ | — | 85 |
| BiKd | 32 | — | 32 | — | 64 |
| BiMETIS | 40 | — | 40 | — | 80 |
| Bi2LevelGrid | $6 \times 6$ | $2 \times 2$ | $6 \times 6$ | $2 \times 2$ | 80 |
| Bi2LevelKd | 32 | 8 | 32 | 8 | 80 |

Table IV.  Characteristics of the Tested Road Networks[a]

| Graph | #nodes | #arcs | Shortest Path | Dijkstra's Algorithm Time [$s$] | Dijkstra's Algorithm #Visited Nodes |
|---|---|---|---|---|---|
| road_network_1 | 362,000 | 920,000 | 250 | 0.26 | 183,509 |
| road_network_2 | 474,000 | 1,169,000 | 440 | 0.27 | 240,421 |
| road_network_3 | 609,000 | 1,534,000 | 580 | 0.30 | 306,607 |
| road_network_4 | 1,046,000 | 2,534,000 | 490 | 0.78 | 522,850 |

[a]The column "shortest paths" provides the average number of nodes on a shortest path.

nodes on a shortest path in the graph. For the unmodified algorithm of Dijkstra, the average running time and number of nodes visited by the algorithm is calculated for 5000 runs. The algorithm stops if the target is reached. For each graph, we generated a demand file with 5000 random shortest path requests so that all algorithms use the same shortest path demands. All running time measurements were made on a single AMD Opteron 2.2 GHz Processor with 8 GB RAM.

All of our experiments were performed with an implementation of the algorithms in C++ using the GCC compiler 3.3. We used the graph data structure from LEDA 4.4 [Mehlhorn and Näher 1999]. In order to measure the unaffected improvement in performance with respect to time, we implemented the algorithms very carefully without using frameworks for reuse of code. Our experiments have shown that using complex class hierarchies with virtual functions increased the time for a single shortest path request by up to a factor of ten (even with aggressive optimization of the compiler).

The algorithm of Dijkstra needs several arrays of node and arc data to store the predecessor arc or the current distance of a node. Since an initialization step at the beginning of each shortest path search would take a long time, we used time stamps: every time a node is visited by the search, it gets the time of the current search. Hence, we know whether the data of the arrays is valid for this search or not. Therefore, the initialization step of all nodes can be omitted. This is also a suitable method in the real-life application—where a central server has to answer a huge number of shortest path queries.
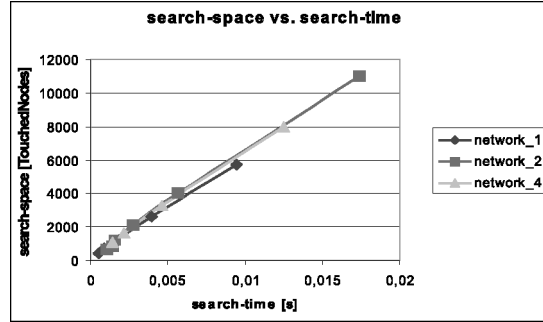
Fig. 8. Search space versus search time of bidirected, accelerated searches. Because of the linearity, it is sufficient to compare the search spaces in further analysis.

Our speedup method reduced the complete graph for each search to a smaller subgraph and this led to a smaller search space. We sampled the average size of the search space by counting the number of nodes that have been visited during a search and we measured the average CPU time per query. A node is "visited" during a search if it is put into the priority queue, i.e., an *Queue.insert* operation is performed.

Dijkstra's algorithm is used as a reference algorithm to compare the search space and CPU time. One reason for the excellent speedup, which can be achieved by the arc-flag technique, is the negligibly small computational overhead that it adds to the standard algorithm of Dijkstra: Dijkstra's algorithm with arc-flags only tests one bit of a bit-vector every time an arc is explored.

Arc-flags can be combined with a bidirectional search. In principle, arc-flags can be used independently for the forward search, the backward search, or both. In our experiments, the best results (with a fixed total number of bits per arc) are achieved through a forward and backward accelerated bidirectional search, which means that we applied the partition-based speedup technique to both search directions (with one-half of the bits for each direction). For the bidirectional search, there are several possible alternation methods for switching between the forward and backward search. In order to reduce the resulting search space, our algorithm takes the direction with the smaller search horizon (the size of the priority queue).

Most of the figures that we present in this paper compare speedups of time or reductions of the search space in relation to the size of preprocessed data, which is the size of the calculated flag-vectors.

## 8. COMPUTATIONAL RESULTS

Figure 8 illustrates that, in the graphs we tested, there is a linear correlation between the search space and the CPU time. This justifies that in the following analysis it is sufficient to consider the search space only.

## 8.1 Unidirectional Search

Figure 9 shows a comparison of the grid and the $kd$-tree partitioning with respect to the number of the partitions. In all graphs we tested, the grid
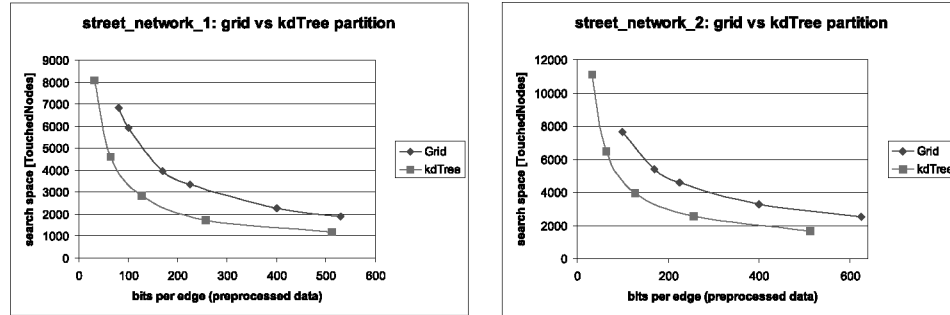
Fig. 9.   Comparison of the quality between grid partitions and $kd$-tree partitions for unidirectional shortest-path searches. The usage of $kd$-tree partitions leads to clearly better results because it adapts to the layout of the graph.

partitioning led to smaller reductions of the search space, because the $kd$ tree adapts better to the 2D layout of the graph. Therefore, we will restrict the following analysis of the partition-based Dijkstra to the $kd$-tree partitions.

Figure 10 shows the absolute and relative size of the search-space, the speedup factor, and the preprocessing time for the unidirectional arc-flag acceleration method using $kd$-tree partitions with increasing sizes of the partitions. The size of the absolute search space clearly depends on the size of the graph. A surprising result is that the relative size of the search space (relative to the search space of the standard algorithm of Dijkstra) is very similar for all graphs, except road_network_3, which seems to have a special characteristic. The speedup diagram of Figure 10 shows the factor between the average time of one accelerated shortest-path search and the running time of the standard algorithm of Dijkstra for the same request.

## 8.2 Two-Level Partitions

The main reason for the introduction of the second-level partitions was that no arc is excluded from the shortest-path search inside the region of the target node $t$. Our experiments have shown that by using the two-level arc-flags the number of nodes that are visited during a search decreases as the search approaches the target node. Figure 11 compares the search spaces of the accelerated searches with one- and two-level arc-flags. Although only very few bits are added, the average search space is reduced to about one-half of its size.

## 8.3 Bidirectional Search

The best results were achieved by a bidirectional Dijkstra search, that is accelerated in both directions: we can measure speedups of more than a factor of 500 compared to the standard algorithm of Dijkstra. In general, the speedup increases with the size of the graph.

Using a bidirectional search, the two-level strategy becomes less important, because the second-level flag-vectors will not be used in most of the shortest path searches: the second-level flag vectors are only used if the search enters the region of the target. During a bidirectional search, the two search horizons are
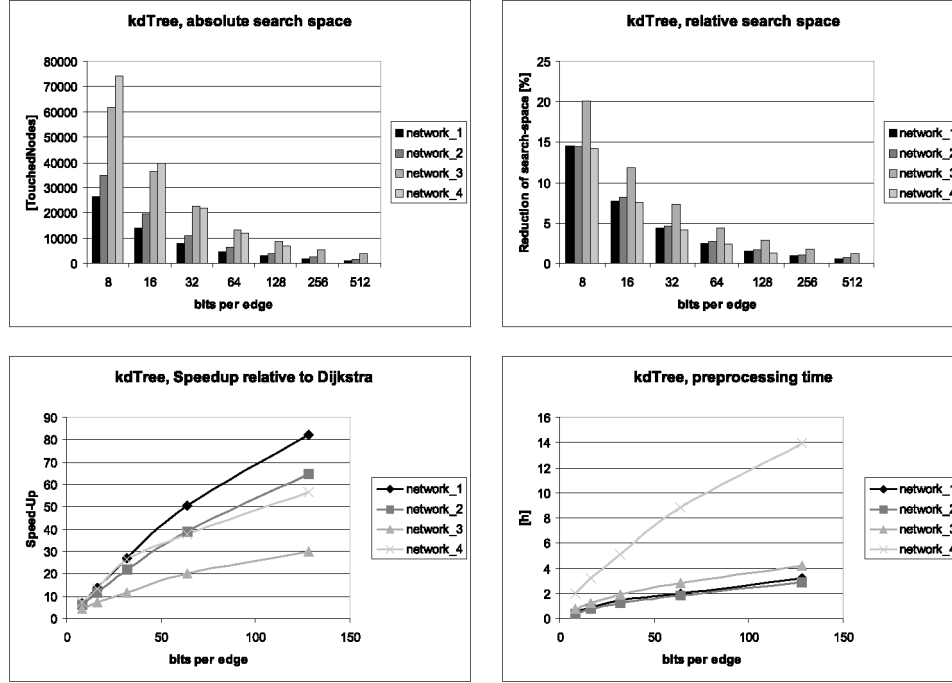
Fig. 10. Results of our unidirectional arc-flag acceleration method using $kd$-tree partitions. The numbers are the average numbers over all performed shortest-path computations. The *absolute search space* is shown by the number of visited or touched nodes during a shortest-path search versus the size of the flag vectors per edge (in bits per edge). The *relative search space* is shown by the reduction of the size of the search space in percentage (the search space of the standard algorithm of Dijkstra equals 100%) versus the size of the flag vectors per edge (in bits per edge). For example, a reduction of the size of the search space to 5% means that the acceleration method visits only 5% of the search space of the standard algorithm of Dijkstra (for the same request) to find a shortest path. The *speedup relative to Dijkstra* shows the absolute speedup factor of the unidirectional arc-flag acceleration method versus the size of the flag vectors per edge (in bits per edge). For example, a speedup of 80 means that the acceleration method finds a shortest path 80 times faster than the standard algorithm of Dijkstra. The *preprocessing time* is shown in hours versus the size of the flag vectors per edge (in bits per edge) for all of our road networks.

likely to meet in a region other than the source or the target region. Therefore, the second-level flag vectors are only used if both nodes are lying in the same region. Figure 12 confirms this estimation: only for large partitions in the first level is a speedup recognizable with two-level flag-vectors. If more than 50 bits for the first level are used, the difference is marginal. Therefore, it does not seem useful to use the second-level strategy in a bidirectional search.

Figure 13 shows the results of the bidirectional search, which is accelerated by arc-flags with $kd$-tree partitions. Even with less preprocessed data (16 bits per arc), we get a speedup of over 50. The accelerated search on network_4 is 545 times faster than the search with the standard algorithm of Dijkstra when using 128 bits per arc preprocessed data (1.3 ms, on average, per accelerated search).
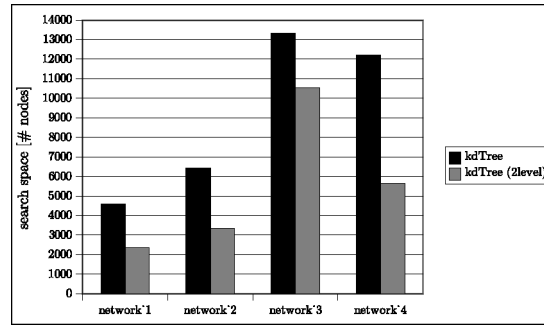
Fig. 11.    Comparison of the search space of the one-level (64 regions) and two-level (64 first-level regions, 8 second-level regions) arc-flags with $kd$ trees.
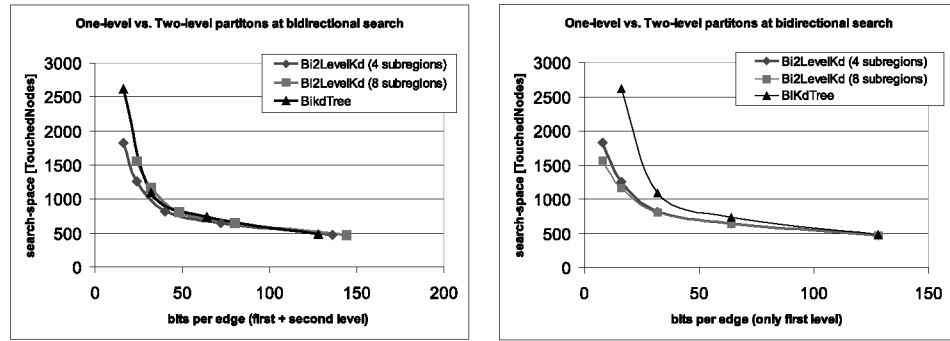


Fig. 12.    Average search space for a bidirectional search using arc-flags with $kd$ trees and a two-versus one-level partition. The y axis shows the size of search space in number of visited (touched) nodes. The x axis of the left figure shows the sum of bits of the arc-flag vectors for the first- and the second-level partition. The x axis of the right figure shows the number of bits only for the arc-flag vector for the first-level partition.
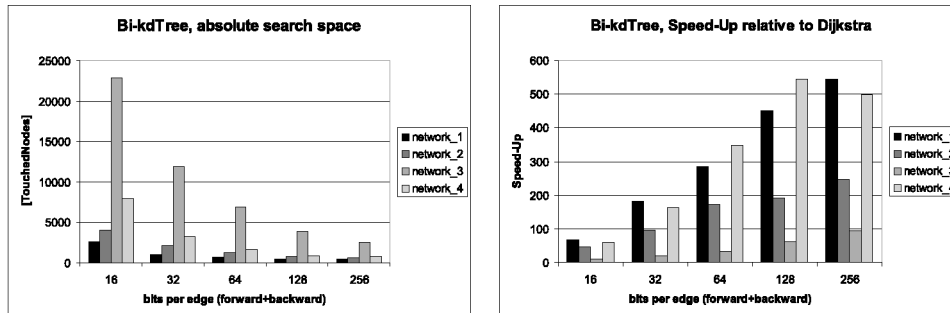


Fig. 13.    Search space and speedup for road networks 1–4 with a bidirectional accelerated search using $kd$-tree partitions.
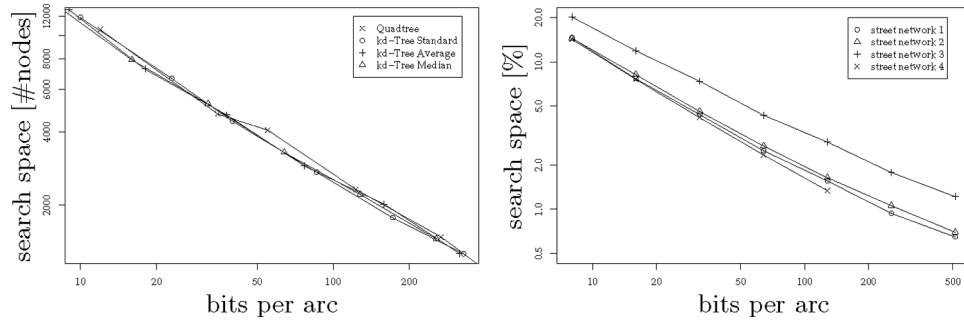
Fig. 14. Average search spaces for different sizes of the flag vectors and unidirectional search. With an increasing number of bits per arc, the search space gets smaller.

## 8.4 Comparison of the Partitioning Methods

We first compared the four geometric partitioning methods quad and $kd$ trees for the center (standard), average, and median. Figure 14a shows the average search space for a road network for an increasing number of bits per arc. As the differences are, indeed, very small, we will use only $kd$ trees with median in the rest of this section as a representative for this partitioning class.

We now compare the average search spaces for different graphs. For an easy comparison, we consider the search space *relative* to the average search space of Dijkstra's algorithm in a given graph. Figure 13b provides the relative average search space for an increasing number of bits per arc. An interesting point is that for arc-flags in this range of size, all curves follow a power law.

Figure 15 compares the results of the different partitioning methods on the four road networks. The size of the preprocessed data per arc is nearly the same for all algorithms: 80 bits. The exact size of 80 bits could not always be realized, as the size of a partition (and therefore the size of the flag-vectors per arc) is determined by specific parameters. The $kd$-tree partitions, for instance, always have a size of powers of two. Table III shows the partitions we used for the comparison.

Figure 12 shows the average search space for a bidirectional search using arc-flags with $kd$ trees for a one- and two two-level partitions with different sizes of the partitions. If more than 50 regions are used for the first-level, the two-level acceleration does not provide any noticeable improvement. Note that in the case of a bidirectional search for a large number of bits per arc the curves are bent, which is the result of some degree of saturation. The same observation can be made in Figure 13, which shows the search space for the four road networks. In contrast, in Figure 14b the curves follow a power law for a unidirectional search.

For the unidirectional searches, the two-level strategies led to the best results: a factor of 2 better than for their corresponding one-level partitioning (see Figure 11). For the bidirectional search, we can see some kind of saturation: the differences between one- and two-level partitions are getting very small with increasing number of bits (see Figure 12).
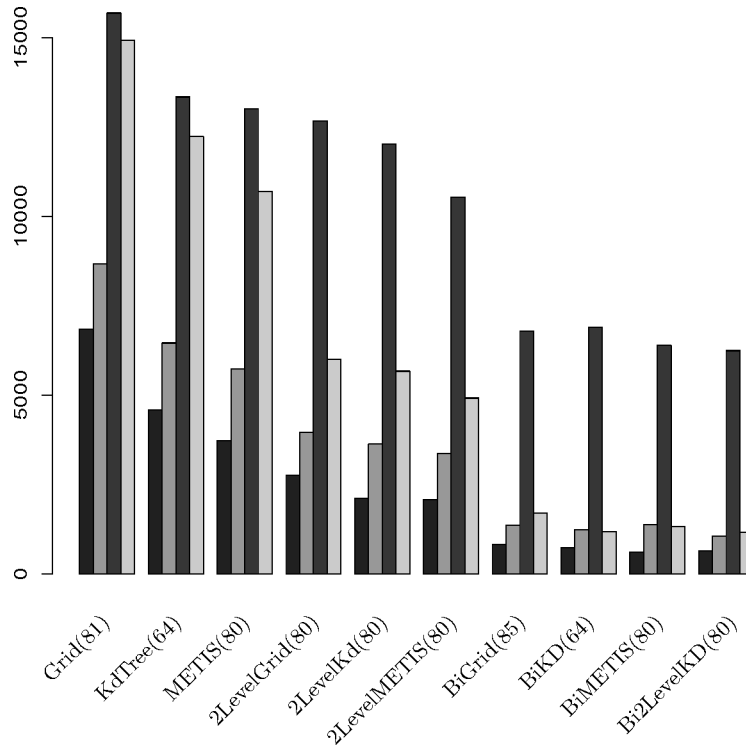
Fig. 15.   Average sizes of the search spaces for most of the implemented algorithms on road networks 1–4 are given on the y axis. The sizes are given in average numbers of visited arcs during a shortest-path search. The sizes of the flag vectors (number of bits per arc) are given in brackets.

In summary, our tests showed that the best results are achieved by the arc-flag method combined with a bidirected search, accelerated in both directions with $kd$-tree or METIS partitions.

## 9. CONCLUSION AND OUTLOOK

For the arc-flag method there is a clear trade-off between speedup 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 (standard algorithm of 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.

Our tests show that the best partition-based speedup methods is a bidirected search, accelerated in both directions with $kd$-tree or METIS partitions. We can measure speedups that are more than 500 times faster than Dijkstra's algorithm. In general, the speedup increases with the size of the graph. The preprocessing effort scales well and can easily be adapted to a parallel algorithm: the flag vector entries of different regions are independent and can be computed in parallel.

Even with the smallest preprocessed data (16 bit per arc), we get a speedup of more than 50. The accelerated search on network_4 is 545 times faster than the standard algorithm of Dijkstra using 128 bits per arc preprocessed data (1.3 ms, on average, per accelerated search). On the basis of our tests, we can recommend the $kd$ tree used for forward and backward acceleration. The partitions with $kd$ trees and METIS yield the highest speedup factors, but $kd$ trees are easier to implement.

It would be particularly interesting to develop a specialized partitioning method that is optimized for the arc-flags approach. Our experiments support our intuition that regions should be of equal size and nodes should be grouped together if the distance between them is small. However, we cannot prove theoretically that this would be the best partitioning method for arc-flags.

One approach for finding good partitions might be the following: consider the flag vectors of the preprocessing of the partitioning with exactly one node per region. Now regard the Hamming distance according to the flag vectors of two nodes—the number of entries that differ for these two nodes over all arcs. These distances form a distance matrix. We can now search a partitioning with $R$ regions where the distances of nodes in one region are minimal according to the distance matrix. This problem is known as *clustering*. Although various other techniques are known for graph clustering, all optimization criteria that we are aware of either result in a large processing time or their use for the arc-flags approach cannot be sufficiently motivated.

For a unidirectional search the two-level arc-flags lead to a significant speedup. The reduction of the search space by far outweighs the overhead to "uncompress" two-level arc-flags. It would be interesting to evaluate whether this effect can be repeated with a third or fourth level of compression, especially for very large graphs like the complete road network of Europe.

There are further known speedup techniques by Sanders and Schultes [2005], Goldberg and Harrelson [2005], Gutman [2004], Holzer [2003], and Jung and Pramanik [1996]. Although the speedup factors of these techniques are not competitive, experimental studies done by Holzer et al. [2004] and Wagner and Willhalm [2005] with similar techniques suggest that combinations outperform a single speedup technique. A systematic evaluation of combinations with current approaches would, therefore, be of great value.

## REFERENCES

AGRAWAL, R. AND JAGADISH, H. V. 1994. Algorithms for searching massive graphs. *IEEE Transactions on Knowledge and Data Engeneering 6*, 2, 225–238.

BATTISTA, G. D. AND ZWICK, U., Eds.  2003.  *Algorithms - ESA 2003, 11th Annual European Symposium, Budapest, Hungary, September 16-19, 2003, Proceedings*. Lecture Notes in Computer Science, vol. 2832. Springer, New York.

CAR, A. AND FRANK, A. U.  1994.  Modelling a hierarchy of space applied to large road networks. In *IGIS '94: Geographic Information Systems, International Workshop on Advanced Information Systems*, J, Nievergelt, T. Roos, H.-J. Schek, and P. Widmayer, Eds. Lecture Notes in Computer Science, vol. 884. Springer, Heidelberg, 15–24.

CHOU, Y.-L., ROMEIJN, H. E., AND SMITH, R. L.  1998.  Approximating shortest paths in large-scale networks with an application to intelligent transportation systems. *INFORMS Journal on Computing 10*, 2, 163–179.

CORMEN, T. H., LEISERSON, C. E., RIVEST, R. L., AND STEIN, C.  2001.  *Introduction to Algorithms*. The MIT Press, Cambridge, MA.

DIJKSTRA, E. W.  1959.  A note on two problems in connexion with graphs. In *Numerische Mathematik*. Vol. 1. Mathematisch Centrum, Amsterdam, The Netherlands, 269–271.

FREDERIKSON, G. N.  1987.  Fast algorithms for shortest paths in planar graphs with applications. *SIAM Journal on Computing 16*, 6, 1004–1022.

FREDMAN, M. L. AND TARJAN, R. E.  1987.  Fibonacci heaps and their uses in improved network optimization algorithms. *Journal of the Association for Computing Machinery 34*, 3, 596–615.

GOLDBERG, A. V. AND HARRELSON, C.  2005.  Computing the shortest path: $A^*$ search meets graph theory. In *Proceedings of the Sixteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, A. Buchsbaum, Ed. SIAM, Philadelphia, PA, USA, 156–165.

GUTMAN, R. J.  2004.  Reach-based routing: A new approach to shortest path algorithms optimized for road networks. In *Proceedings of the Sixth Workshop on Algorithm Engineering and Experiments and the First Workshop on Analytic Algorithmics and Combinatorics*, L. Arge, G. F. Italiano, and R. Sedgewick, Eds. SIAM, Philadelphia, PA, USA, 100–111.

HOLZER, M.  2003.  Hierarchical speedup techniques for shortest path algorithms. Tech. rep., Dept. of Informatics, University of Konstanz, Germany.

HOLZER, M., SCHULZ, F., AND WILLHALM, T.  2004.  Combining speedup techniques for shortest-path computations. In *Experimental and Efficient Algorithms, Third International Workshop*, C. C. Ribeiro and S. L. Martins, Eds. Lecture Notes in Computer Science, vol. 3059. Springer, Heidelberg, 269–284.)

JOHNSON, D. B.  1977.  Efficient algorithms for shortest paths in sparse networks. *Journal of the Association for Computing Machinery 24*, 1, 1–13.

JUNG, S. AND PRAMANIK, S.  1996.  Hiti graph model of topographical roadmaps in navigation systems. In *Proceedings of the Twelfth International Conference on Data Engineering*, S. Y. W. Su, Ed. IEEE Computer Society, Tokyo, 76–84.

KARYPIS, G.  1995.  METIS: Family of multilevel partitioning algorithms. `http://www-users.cs.umn.edu/~karypis/metis/`.

KARYPIS, G. AND KUMAR, V.  1998.  A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM Journal on Scientific Computing 20*, 1 (Aug.), 359–392.

KÖHLER, E., MÖHRING, R. H., AND SCHILLING, H.  2005.  Acceleration of shortest path and constrained shortest path computation. In *Experimental and Efficient Algorithms, 4th InternationalWorkshop, WEA 2005*, S. E. Nikoletseas, Ed. Lecture Notes in Computer Science, vol. 3503. Springer, Heidelberg, 126–138.

KRANAKIS, E., KRIZANC, D., AND URRUTIA, J.  1995.  Implicit routing and shortest path information (extended abstract). In *SIROCCO*, L. M. Kirousis and C. Kaklamanis, Eds. Proceedings in Informatics, vol. 2. Carleton Scientific, 101–112.

LAUTHER, U.  1997.  Slow preprocessing of graphs for extremely fast shortest path calculations. Lecture at the Workshop on Computational Integer Programming at ZIB.

LAUTHER, U.  2004.  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*, M. Raubal, A. Sliwinski, and W. Kuhn, Eds. IfGI prints, vol. 22. Institut für Geoinformatik, Westfälische Wilhelms-Universität, Münster, 219–230.

MEHLHORN, K. AND NÄHER, S.  1999.  *LEDA, A platform for Combinatorial and Geometric Computing*. Cambridge University Press, Cambridge.

SANDERS, P. AND SCHULTES, D. 2005. Highway hierarchies hasten exact shortest path queries. In *Algorithms—ESA 2005, 13th Annual European Symposium*, G. S. Brodal and S. Leonardi, Eds. Lecture Notes in Computer Science, vol. 3669. Springer, 568–579.

SCHULZ, F. 2005. Timetable information and shortest paths. Ph.D. thesis, Faculty of Informatics, University of Karlsruhe.

SCHULZ, F., WAGNER, D., AND WEIHE, K. 2000. Dijkstra's algorithm on-line: An empirical case study from public railroad transport. *ACM Journal of Experimental Algorithms 5*, 12.

WAGNER, D. AND WILLHALM, T. 2003. Geometric speedup techniques for finding shortest paths in large sparse graphs. (See Battista and Zwick [2003], 776–787.)

WAGNER, D. AND WILLHALM, T. 2005. Drawing graphs to speed up shortest-path computations. In *Proceedings of the Seventh Workshop on Algorithm Engineering and Experiments and the Second Workshop on Analytic Algorithmics and Combinatorics*, C. Demetrescu, R. Sedgewick, and R. Tamassia, Eds. SIAM, Philadelphia, PA, USA, 17–25.

WAGNER, D., WILLHALM, T., AND ZAROLIAGIS, C. 2005. Geometric containers for efficient shortest path computation. *ACM Journal of Experimental Algorithms 10*.

WILLHALM, T. 2005. Engineering shortest paths and layout algorithms for large graphs. Ph.D. thesis, Faculty of Informatics, University of Karlsruhe.