Partitioning Graphs to Speed Up Dijkstra's Algorithm

Rolf H. Möhring[1], Heiko Schilling[1], Birk Schütz[2], Dorothea Wagner[2], and Thomas Willhalm[2]

[1] Technische Universität Berlin, Institut für Mathematik, Straße des 17. Juni 136, 10623 Berlin, Germany.
[2] Universität Karlsruhe, Fakultät für Informatik, Postfach 6980, 76128 Karlsruhe, Germany.

Abstract. In this paper, we consider Dijkstra's algorithm for the point-to-point shortest path problem in large and sparse graphs with a given layout. In [1], a method has been presented that uses a partitioning of the graph to perform a preprocessing which allows to speed-up Dijkstra's algorithm considerably.
We present an experimental study that evaluates which partitioning methods are suited for this approach. In particular, we examine partitioning algorithms from computational geometry and compare their impact on the speed-up of the shortest-path algorithm. Using a suited partitioning algorithm speed-up factors of 500 and more were achieved.
Furthermore, we present an extension of this speed-up technique to multiple levels of partitionings. With this multi-level variant, the same speed-up factors can be achieved with smaller space requirements. It can therefore be seen as a compression of the precomputed data that conserves the correctness of the computed shortest paths.

1 Introduction

We consider the problem of repeatedly finding shortest paths in a large but sparse graph with given arc weights. Dijkstra's algorithm [2] solves this problem efficiently with a sub-linear running time as the algorithm can stop once the target node is reached. If the graph is static, a further reduction of the search space can be achieved with a preprocessing that creates additional information. More precisely, we consider the approach to enrich the graph with arc labels that mark, for each arc, possible target nodes of a shortest paths that start with this arc. Dijkstra's algorithm can then be restricted to arcs whose label mark the target node of the current search, because a sub-path of a shortest path is also a shortest path.

This concept has been introduced in [3] for the special case of a timetable information system. There, arc labels are angular sectors in the given layout of the train network. In [4], the approach has been studied for general weighted graphs. Instead of the angular sectors, different types of convex geometric objects are implemented and compared.

A different variation has been presented in [1], where the graph is first partitioned into regions. Then an arc-flag then consists of a bit-vector that marks the regions containing target nodes of shortest paths starting with this arc. Usually, arc-flags result in a much smaller search space for the same amount of preprocessed data. Furthermore, the generation of arc-flags can be realized without the computation of all-pairs shortest paths in contrast to the geometric objects of [4]. In fact, only the distances to nodes are needed that lie on the boundary of a region. (See [1] for details.) However in [5], it has been shown that partitioning of the graph with METIS [6] generally results in a better reduction than for the partitioning algorithms of [1].

The first contribution of this paper is a computational study whether partitioning algorithms from computational geometry can be used for the arc-flag approach and how they compare to the results of METIS. The algorithms are evaluated with large road networks, the typical application for this problem.

As a second contribution of this paper, we present a multi-level version of arc-flags that produces the same speed-up with lower space consumption. Therefore, these multi-level arc-flags can be seen as a (lossy) compression of arc-flags. Note that the compression still guarantees the correctness of a shortest-path query but may be slower than the uncompressed arc-flags.

We start in Sect. 2 with some basic definitions and a precise description of the problem and the pruning of the search space of Dijkstra's algorithm with arc-flags. In Sect.3, we present the selection of geometric partitioning algorithms that we used for our analysis. We discuss the two-level variant of the arc-flags in Sect. 4. In Sect. 5, we describe our experiments and we discuss the results of the experiments in Sect. 6. We conclude the paper with Sect. 7.

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 $u_1, \dots, u_k$ 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 $m \in O(n)$. 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 Shortest Path Problem

Let $G = (V, A)$ be a directed graph whose arcs are *weighted* by a function $l : A \to \mathbb{R}$. 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 [7], to convert in $O(nm + n^2 \log n)$ time the original arc weights $l : A \to \mathbb{R}$ to non-negative arc weights $l' : A \to \mathbb{R}_0^+$ that result in the same shortest paths. Hence, in the rest of the paper, we can safely assume that arc weights are non-negative. 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.[3]

2.3 Dijkstra's Algorithm with Arc-Flags

The classical algorithm for computing shortest paths in a directed graph with non-negative arc weights is that of Dijkstra [2]. For the general case of arbitrary non-negative arc lengths, it still seems to be the fastest algorithm with $O(m + n \log n)$ worst-case time. However, in practice, speed-up techniques can reduce the running time and often result in a sub-linear 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 one admits a preprocessing, 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 sub-graph with those arcs a for which the target t is in $S(a)$. However, storing all sets $S(a)$ requires $O(nm)$ space which results in $O(n^2)$ space for sparse graphs with $m \in O(n)$ and is thus prohibitive 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 $r : V \to \{1, \dots, p\}$ that assigns to each node the number of its region. (Given a 2D layout of the graph, a simple method to partition a graph is to use a regular grid as illustrated in Figure 1 and assign all nodes inside a grid cell the same number.) We will now use a bit-vector $b_a : \{1, \dots, p\} \to \{\texttt{true}, \texttt{false}\}$ with p bits, each of which corresponds to a region. For each arc a, we therefore set the bit $b_a(i)$ to `true` iff a is the beginning of a shortest path to at least one node in region $i \in \{1, \dots, p\}$. For a specific shortest-path query

[3] This can be achieved by adding a small fraction to the arc weights, if necessary.

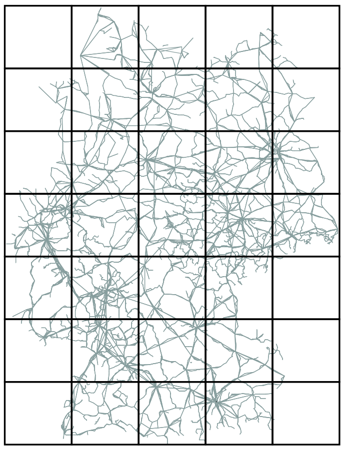


Fig. 1. A 5×7 grid partitioning of Germany

from s to t, Dijkstra's algorithm can be restricted to the sub-graph G_t with those arcs a for which the bit of the target-region is set to `true`. (For all edges on the shortest path from s to t the arc-flag for the target region is set, because a sub-path of a shortest path is also a shortest path.)

The sub-graph G_t can be computed on-line during Dijkstra's algorithm. In a shortest-path search from s to t, while scanning a node u, the modified algorithm considers all outgoing arcs but ignores those arcs which have not set the bit for the region of the target node t. All possible partitions of the nodes lead to a correct solution but most of them would not lead to the desired speed-up of the computation.

The space requirement for the preprocessed data is $\mathcal{O}(p \cdot m)$ for p regions because we have to store one bit 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 regarding unnecessary arcs or nodes. Note however, that in practice even for $p \ll n$ we achieved an average search space that is only 4 times the number of nodes in the shortest path. Furthermore, it is possible within the framework of the arc-flag speed-up 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.)

3 Partitioning Algorithms

The arc-flag speed-up 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 and the algorithm returns a shortest path, but most partitions do not lead to an acceleration. In this section, we will present the partitioning algorithms that we examined. Most of these algorithms need a 2D layout of the graph.

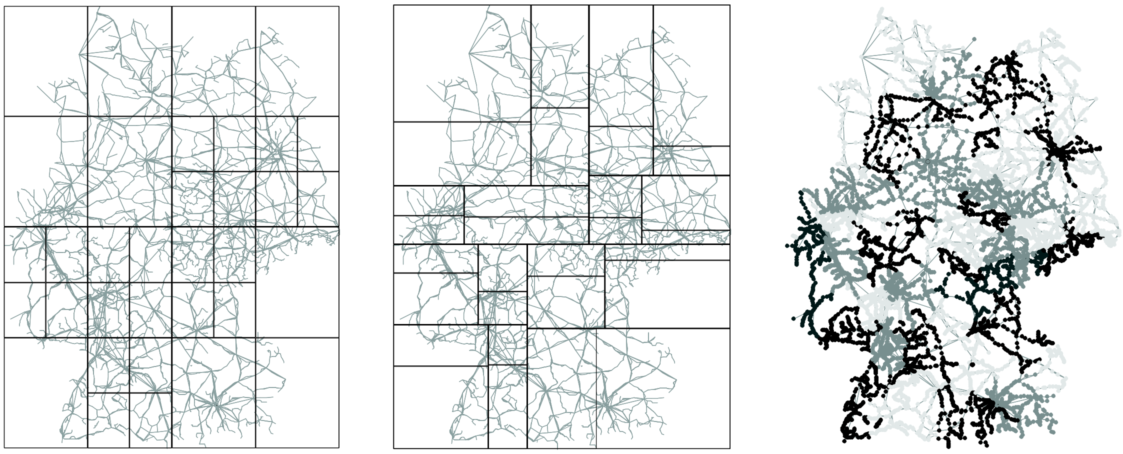
3.1 Grid

Probably the easiest way to partition a graph with a 2D layout is to use regions induced by a grid of the bounding box. Each grid cell defines one region of the graph. Nodes on a grid line are assigned to an arbitrary but fixed grid cell. Figure 1 shows an example of a 5×7 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 in [4] approximate $S(u,v)$ by a single convex object of constant size.)

The grid partitioning method uses only 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 the grid partitioning always has the worst results in our experiments. Since [1, 5] include this partitioning method, we use the grid partitioning as a baseline and compare all other partitioning algorithms with it.

3.2 Quadtrees



(a) Quadtree (34 regions) (b) Kd-Tree (32 regions) (c) METIS (32 regions).

Fig. 2. Germany with three different partitions

A *quadtree* is a data structure for storing points in the plane. Quadtrees are typically used in computational geometry for range queries and have applications in computer graphics, image analysis, and geographic information systems.

Definition 1 (Quadtree). *Let P be a set of points in the plane and R_0 its bounding-box. Then, the data structure* quadtree *is a rooted tree of rectangles, where*

– *the root is the bounding region R_0, and*
– *R_0 and all other regions R_i are recursively divided into the four quadrants, while they contain more than one point of P.*

The leaves of a quadtree form a subdivision of the bounding-box R_0. Even more, the leaves of every sub-tree 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 sub-tree of the quadtree. More precisely, we define an upper bound $b \in \mathbb{N}$ of points in a region and stop the division if a region contains less points than this bound b. This results in a partition of our graph where each region contains at most b nodes. Fig. 2(a) shows such a partition with 34 regions. In contrast to the grid-partition, this partitioning reflects the geometry of the graph—dense parts will be divided into more regions than sparse parts.

3.3 Kd-Trees

In the construction of a quadtree, a region is recursively divided into four equally-sized sub-regions. However, equally-sized sub-regions do not take into account the distribution of the points. This leads to the definition of a *kd-tree*. In the construction of a *kd-tree*, the plane is recursively divided in a similar way as for a quadtree. The underlying rectangle is decomposed into *two halves* by a straight line parallel to an axis. The directions of the dividing line alternate. 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. (Fig. 2(b) shows a result for the median and 32 regions.) If the median of points in general position is used, the partitioning has always 2^l regions.

The median of the nodes can be computed in linear time with the *median of medians* algorithm [8]. Since the running time of the preprocessing is dominated by the shortest-path computations after the partitioning of the graph, we decided to use a standard sorting algorithm instead. (As a concrete example, the *kd-tree* partitioning with 64 regions for one of our test graphs with one million nodes was calculated in 175s, calculating the arc-flags took seven hours.)

3.4 METIS

A fast method to partition a graph into k almost equally-sized sets with a small cut-set is presented in [9]. An efficient implementation can be obtained free-of-charge from [6]. 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 cut is noticeable smaller than in the other partitioning methods. Fig. 2(c) shows a partitioning of a graph generated by METIS.

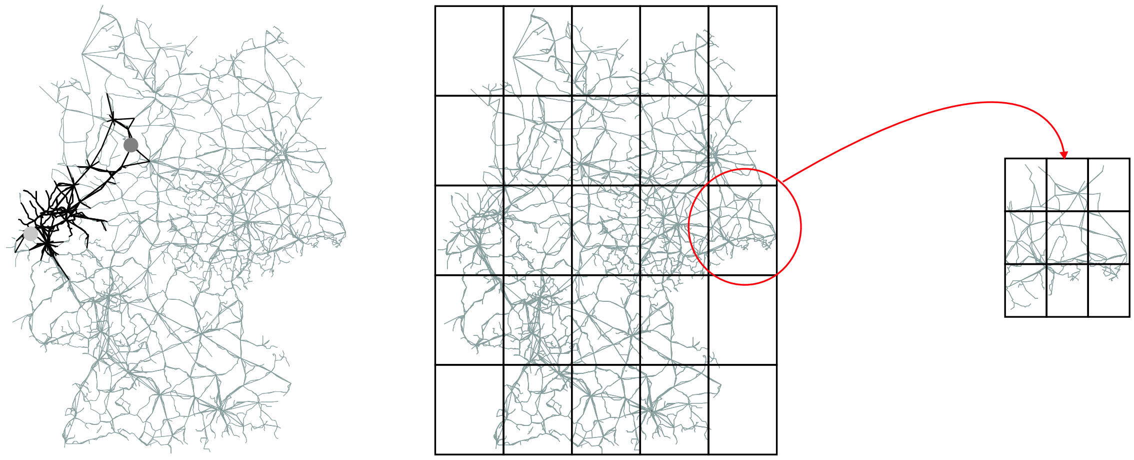
4 Two-Level Arc-Flags

graph	#arcs	algorithm	# marked regions		
			$=1$	$<10\%$	$>95\%$
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

Table 1. Some statistics about the number of regions that are marked in arc-flags.

An analysis of the calculated arc-flags reveals that there might exist possibilities to compress the arc-flags. For 80% of the arcs either almost none or nearly all bits of their arc-flags are set. Table 1 shows an excerpt of the analysis we made. The column "$= 1$" shows the number of arcs, which are only responsible for shortest paths inside their own region (only one bit is set). Arcs with more than 95% bits set could be important roads. This justifies ideas for (lossy) compression of the arc-flags, but it is important that the decompression algorithm is very fast—otherwise the speed-up of time will be lost.

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 3(a) for a search from the dark grey node to the light grey node, the modified DIJKSTRA search reduces the search space next to 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 you consider that usually all arcs of a region have set the region-bit of their own region. We could handle this problem if we used a finer partition of the graph but this would lead to longer arc-flags



(a) Without two-level arc-flags a search visits almost all arcs in the target region.

(b) For each arc a bit-vector is stored for the coarse 5×5 grid and a bit-vector for a fine 3×3 grid *in the same coarse region as the arc.*

Fig. 3. Illustrations for two-level vectors

(requiring more memory and a longer preprocessing). Take the following example, if we used a 15×15 grid instead of a 5×5 grid, each region would be split in 9 additional regions but the preprocessing data increases from 25 to 225 bits per arc. However, the additional information for the fine grid is mainly needed for 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 this additional data (for the fine grid) only for the arcs inside the same coarse region. Therefore, each arc gets two bit-vectors: one for the coarse partition and one for the associated region of the fine partition.

The advantage of this method is that the preprocessed data is smaller than for a fine one-level partitioning, because the second bit-vector exists only for the target region (34 bits per arc instead of 225). It is clear that the 15×15 grid would lead to better results. However, the difference for the search spaces is small because we expect that entries in arc-flags of neighboring regions are similar for regions far away. Thus, we can see this two-level method as a compression of the first-level arc-flags. We summarize the bits for remote regions. If one bit is set for a fine region, the bit 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 arc-flags of level one. If the algorithm has entered the target region, the second-level arc-flags provide further information on whether an arc can be ignored for the search of a shortest path to the target-node.

Experiments showed (Section 6) that this method leads to the best results concerning the reduction of the search space, but an increased preprocessing effort is needed. Note however, that it is not necessary in the preprocessing 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 are finished.

5 Experimental Setup

The main goal of this section is to compare the different partitioning algorithms with regard to their resulting search space and speed-up of time during the accelerated DIJKSTRA search. We tested the algorithms on German road networks, which are directed and have a 2D layout and positive arc weights. Table 2 shows some characteristics of the graphs. The column "shortest path"

Graph	#nodes	#arcs	shortest path	Dijkstra's algorithm time [s]	Dijkstra's algorithm #touched 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

Table 2. Characteristics of tested road networks. The columns "shortest paths" provides the average number of nodes on a shortest path.

is the average number of nodes on a shortest path in the graph. For the unmodified Dijkstra's algorithm, the average running time and number of nodes touched by the algorithm is given for 5000 runs.

All experiments are 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 [10]. As we do not have real-world shortest-path queries we considered random queries for our experiments. 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 runtime measurements were made on a single AMD Opteron Processor with 2.2 GHz and 8 GB RAM. Note that our algorithms are not optimized with respect to space consumption. However, even for the largest graphs considered in this study significant less than 8 GB RAM would suffice.

Our speed-up method reduces the complete graph for each search to a smaller sub-graph and this leads to a smaller search space. We sampled the average size of the search space by counting the number of visited nodes and measured the average CPU time per query. Dijkstra's algorithm is used as a reference algorithm to compare search space and CPU time. Fortunately, Dijkstra's algorithm with arc-flags only tests a bit of a bit-vector and does not lead to a significant overhead. In graphs we tested, there is a strong linear correlation between the search space and the CPU time. This justifies that in the analysis it is sufficient to consider the search space only.

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 of them. In our experiments the best results (with a fixed total number of bits per arc) achieved a forward and backward accelerated bidirectional search, which means that we applied the partition-based speed-up technique on both search directions (with half of the bits for each direction).

6 Computational Results

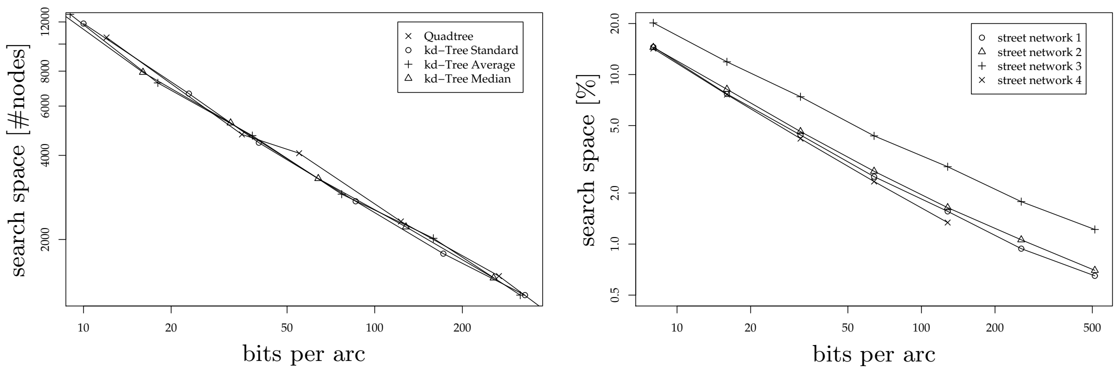
6.1 Quadtrees and Kd-Trees

We first compared the four geometric partitioning methods quadtrees and kd-trees for the center (standard), average, and median. Figure 4(a) 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 space for different graphs. For an easy comparison we consider the search space *relative* to the average search space of Dijkstra's algorithm in this graph. Figure 4(b) provides the relative average search space for an increasing number of bits per arc. It is remarkable that for arc-flags in this range of size all curves follow a power law.

6.2 Two-Level Partitionings

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. Therefore, the second-level arc-flags reduce the shortest-path search mainly if the search already approaches the target. Figure 5



(a) Partitioning with quadtree and three kd-tree partitions (standard, average, and median). The difference for the resulting search space is marginal.

(b) Partitioning with median kd-tree for road network 1-4. The search space is plotted relative to the search space of Dijkstra's algorithm.

Fig. 4. Average search space for different sizes of arc-flags. With an increasing number of bits per arc, the search space gets smaller.

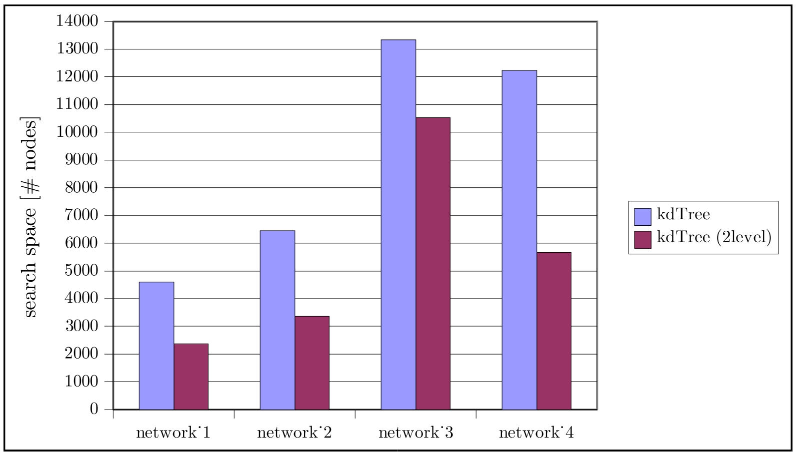


Fig. 5. Comparison of one-level (64 regions) and two-level (64 first-level regions, 8 second-level regions) arc-flags with kd-trees.

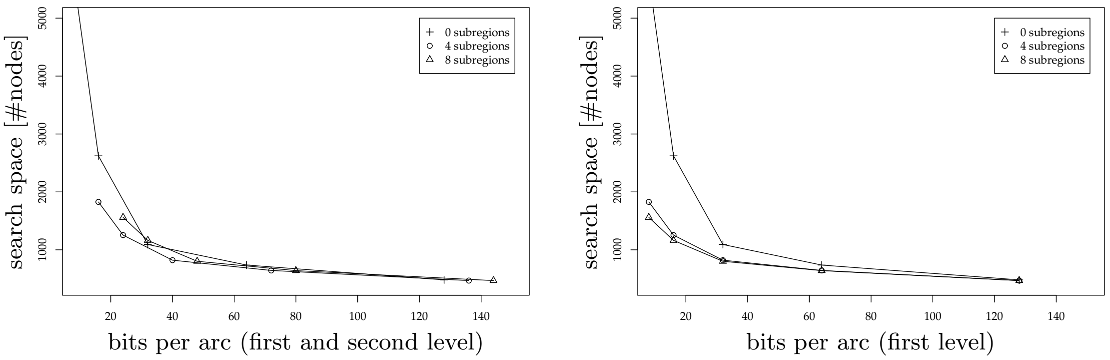


Fig. 6. Average search space for a bidirectional search using arc-flags by kd-trees. The two-level strategy becomes irrelevant for the bidirectional search. If more than 50 regions are used for the first-level, the two-level acceleration does not provide any noticeable improvement.

compares the search spaces of the one-level and two-level accelerated searches. Although only very few bits are added, the average search space is reduced to about half of its size.

Using a bidirectional search, the two-level strategy becomes less important, because the second-level arc-flags will not be used in most of the shortest-path searches: the second-level arc-flags are only used, if the search enters the region of the target. During a bidirectional search the probability is high that the two search horizons meet in a different region than the source or target region. Therefore, the second-level arc-flags are mainly used, if both nodes are lying in the same region. Figure 6 confirms this estimation. Only for large partitions in the first level is a speed-up recognizable. If more than 50 bits for the first level are used, the difference is very small. We conclude that the second-level strategy does not seem to be useful in a bidirectional search.

6.3 Comparison of the Partitioning Methods

Finally, we want to compare the different algorithms directly. We have four orthogonal dimensions in our algorithm tool-box:

1. The base partitioning method: Grid, KdTree or METIS
2. The number of partitions
3. Usage of one-level partitions or two-level partitions
4. Unidirectional or bidirectional search

Since computing all possible combinations on all graphs takes way too much time, we selected the algorithms that are listed in Table 3. (We refrained from implementing Bi2Metis, because usually the two-level arc-flags in a bidirectional search hardly performed better than the one-level variant.) Furthermore, we fix the size of the preprocessed data to nearly the same number for all algorithms. (The same size can not be realized due to the restrictions by the construction of the partitioning algorithms.) Figure 7 compares the results of our partitioning methods on the four road networks.

For the unidirectional searches, the two-level strategies yield the best results (a factor of 2 better than for their corresponding one-level partitioning). For the bidirectional search, we can see some kind of saturation: the differences between the partitioning techniques are very small.

Figure 8 shows the search space for the four road networks. Note that in the case of a bidirectional search, a large number of bits per arc already shows some effects of saturation as the curves are bent. In contrast, in Fig. 4(b) the curves follow a power-law.

Name of partitioning	forward		backward		bits per arc
	1^{st} level	2^{nd} level	1^{st} level	2^{nd} level	
Grid	9×9	-	-	-	81
KdTree	64	-	-	-	64
METIS	80	-	-	-	80
2LevelGrid	8×8	4×4	-	-	80
2LevelKd	64	16	-	-	80
2LevelMETIS	72	8	-	-	80
BiGrid	7×7	-	6×6	-	85
BiKd	32	-	32	-	64
BiMETIS	40	-	40	-	80
Bi2LevelGrid	6×6	2×2	6×6	2×2	80
Bi2LevelKd	32	8	32	8	80

Table 3. Partitionings with nearly the same preprocessed data size of 80 bit

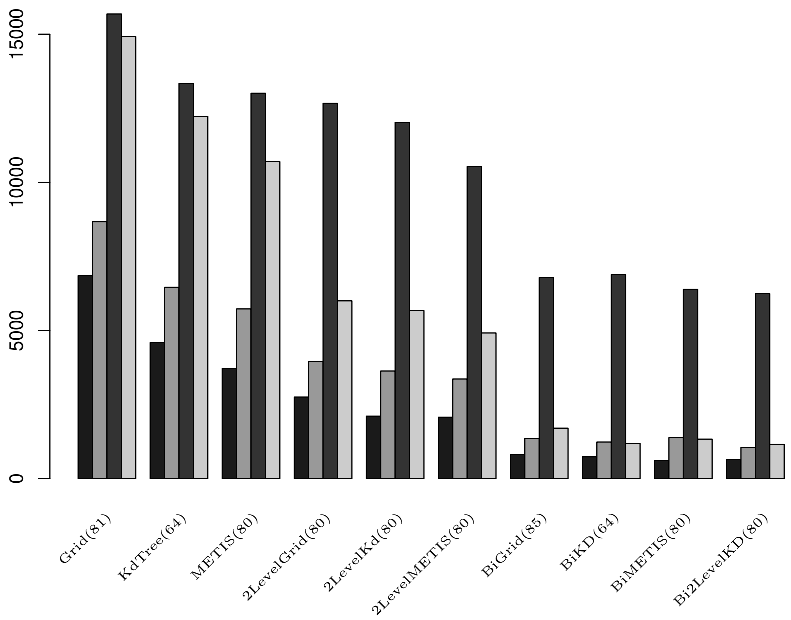


Fig. 7. Average search space for most of the implemented algorithms in road networks 1-4. The number of bits are noted in brackets.

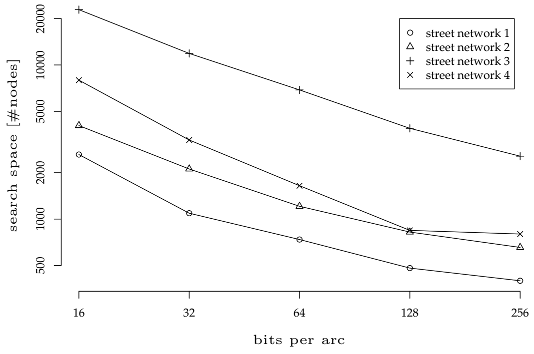


Fig. 8. Search space for road networks 1-4 with a bidirectional accelerated search using kd-tree partitions.

7 Conclusion and Outlook

The best partition-based speed-up method we tested is a bidirectional search, accelerated in both directions with kd-tree or METIS partitions. We can measure speed-ups of more than 500. (In general, the speed-up increases with the size of the graph.) Even with the smallest preprocessed data (16 bit per arc), we get a speed-up of more than 50. The accelerated search on network_4 is 545 times faster than plain Dijkstra's algorithm using 128 bits per arc preprocessed data (1.3ms per search). Of the tested partitioning methods, we can recommend the kd-tree used for forward and backward acceleration. The partitionings with kd-trees and METIS yield the highest speed-up 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 showed that our intuition is right that regions should be equally sized and nodes should be grouped together if their graph-theoretic distance is small. However, we cannot prove that our intuition is theoretically the best partitioning method for arc-flags. Although many further techniques are known from graph clustering, all optimization criteria that we are aware of either result in a large running time or their use for the arc-flags approach cannot be motivated.

For an unidirectional search the two-level arc-flags lead to a considerable speed-up. The reduction of the search space outweighs by far the overhead to "uncompress" two-level arc-flags. It would, however, 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 speed-up techniques [11–14]. Although the speed-up factors of these speed-up techniques are not competitive, experimental studies [15, 16] with similar techniques suggest that combinations outperform a single speed-up technique. A systematic evaluation of combinations with current approaches would therefore be of great value.

References

1. Lauther, U.: An extremely fast, exact algorithm for finding shortest paths in static networks with geographical background. In Raubal, M., Sliwinski, A., Kuhn, W., eds.: Geoinformation und Mobilität - von der Forschung zur praktischen Anwendung. Volume 22 of IfGI prints., Institut für Geoinformatik, Münster (2004) 219–230
2. Dijkstra, E.W.: A note on two problems in connexion with graphs. Numerische Mathematik **1** (1959) 269–271

3. Schulz, F., Wagner, D., Weihe, K.: Dijkstra's algorithm on-line: An empirical case study from public railroad transport. ACM Journal of Experimental Algorithmics **5** (2000)
4. Wagner, D., Willhalm, T.: Geometric speed-up techniques for finding shortest paths in large sparse graphs. In Battista, G.D., Zwick, U., eds.: Proc. 11th European Symposium on Algorithms (ESA 2003). Volume 2832 of LNCS., Springer (2003) 776–787
5. Köhler, E., Möhring, R.H., Schilling, H.: Acceleration of shortest path and constrained shortest path computation. In Nikoletseas, S.E., ed.: WEA 2005: 4th International Workshop on Efficient and Experimental Algorithms. Volume 3503 of LNCS., Heidelberg, Springer (2005) 126–138
6. Karypis, G.: METIS: Family of multilevel partitioning algorithms. http://www-users.cs.umn.edu/ karypis/metis/ (1995)
7. Johnson, D.B.: Efficient algorithms for shortest paths in sparse networks. Journal of the ACM (JACM) **24** (1977) 1–13
8. Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: Introduction to Algorithms. The MIT Press, Cambridge Massachusetts (2001)
9. Karypis, G., Kumar, V.: A fast and high quality multilevel scheme for partitioning irregular graphs. SIAM Journal on Scientific Computing archive **20** (1998) 359–392
10. Mehlhorn, K., Näher, S.: LEDA, A platform for Combinatorial and Geometric Computing. Cambridge University Press (1999)
11. Jung, S., Pramanik, S.: HiTi graph model of topographical road maps in navigation systems. In: Proc. 12th IEEE Int. Conf. Data Eng. (1996) 76–84
12. Holzer, M.: Hierarchical speed-up techniques for shortest-path algorithms. Technical report, Dept. of Informatics, University of Konstanz, Germany (2003) http://www.ub.uni-konstanz.de/kops/volltexte/2003/1038/.
13. Goldberg, A.V., Harrelson, C.: Computing the shortest path: a^* search meets graph theory. Technical Report MSR-TR-2004-24, Microsoft Research (2003) Accepted at SODA 2005.
14. Gutman, R.: Reach-based routing: A new approach to shortest path algortihms optimized for road networks. In Arge, L., Italiano, G.F., Sedgewick, R., eds.: Proc. Algorithm Engineering and Experiments (ALENEX'04), SIAM (2004) 100–111
15. Holzer, M., Schulz, F., Willhalm, T.: Combining speed-up techniques for shortest-path computations. In Ribeiro, C.C., Martins, S.L., eds.: Experimental and Efficient Algorithms: Third International Workshop, (WEA 2004). Volume 3059 of LNCS., Springer (2004) 269–284
16. Wagner, D., Willhalm, T.: Drawing graphs to speed up shortest-path computations. In: Proc. 7th Workshop Algorithm Engineering and Experiments (ALENEX'05). LNCS, Springer (2005) To appear.