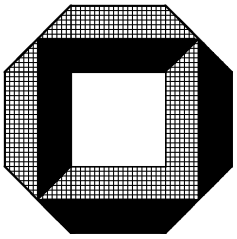
Partition-Based Speed-Up of Dijkstra's Algorithm



Studienarbeit am Institut für Logik, Komplexität und Deduktionssysteme
Lehrstuhl Prof Dr. Dorothea Wagner
Universität Karlsruhe (TH)
Fakultät für Informatik

von
Birk Schütz

Betreuer:
Thomas Willhalm

Abstract

Determining the shortest path from one node to another in a graph is probably the most popular question in graph theory. If the graph is non-negatively weighted, DIJKSTRA'S ALGORITHM is the classic algorithm used to answer this question. Because of its breadth-first-search character, this algorithm usually spreads circularly around the source node of the search and hence the search space can be very large. For the application of dealing with huge numbers of shortest-path queries in static graphs, we consider an algorithm, which uses preprocessed data to decrease the search space for each shortest-path request. The algorithm partitions the graph and, for each edge, the preprocessing considers the relevant regions which have the shortest path over this edge. We will see that the preprocessing scales well and usually runs in almost linear time.

This document shows experimental results for several partitioning algorithms resulting in smaller search spaces on real-world street networks. The quality of these strategies will be compared. A two-level kd-tree with bidirectional search delivered the smallest search space for DIJKSTRA'S ALGORITHM for most of the tested street networks.

This paper was presented as "Studienarbeit" at the university of Karlsruhe (TH) at the institute of Prof. Dr. D. Wagner. I would like to thank Thomas Willhalm and Heiko Schilling for this interesting topic, their ideas and help.

Hiermit versichere ich, dass ich die vorliegende Arbeit selbständig angefertigt habe und nur die angegebenen Hilfsmittel und Quellen verwendet wurden.

Karlsruhe, im November 2004

Contents

1 Introduction

We consider a typical application in traffic systems where a central server has to answer a huge number of customer queries asking for their best itineraries (e.g., a route planning server for cars or a timetable server of a railway provider).

The algorithmic core problem that underlies these applications is the single-source shortest-path problem on directed, non-negatively weighted graphs. DIJKSTRA'S ALGORITHM is the classical method for solving this problem. It can be seen as a weighted breadth-first-search and hence the search space for shortest paths usually spreads circularly around the source and is very large.

Here we consider an accelerating method for DIJKSTRA'S ALGORITHM by exploiting a characteristic of above applications: The underlying data network does not change for a certain period of time. This justifies a preprocessing of the network to speed-up the single queries. Although a computation of the shortest paths for all pairs of nodes would lead to a constant-time response, the quadratic space requirement is intolerable for large graphs with millions of nodes and more.

There are several techniques of speeding up DIJKSTRA'S ALGORITHM which do not use any preprocessed data like the $A*$-algorithm, which is also known as *goal-directed search* or the bidirectional search which searches the shortest path in both directions, from the source to the target and vice versa. The latter can easily be combined with other speed improvement methods. Also for the preprocessed approach, several acceleration techniques exist, for example *angular sectors* [SWW00], *multi-level search* [SWZ02] and *geometric shortest-path containers* [WWZ04].

In this paper, we consider another preprocessing-based acceleration technique which scales fairly well, meaning that it is also suitable for large networks. The preprocessed data can be stored and calculated in $O(k \cdot n \log n)$ for sparse graphs where k is a small constant. Experiments show that the number of nodes visited by DIJKSTRA'S ALGORITHM can be reduced to less than 0,2% (Figure 1 illustrates this situation). We achieve that by ignoring edges in our search of which we know, that they can not lead via a shortest path to the requested target. More precisely, we partition the nodes of the graph and calculate in the preprocessing for each edge those partitions to which a shortest path over this edge exists. While running DIJKSTRA'S ALGORITHM we look up for every edge this information and if the target node is not in a shortest-path partition of the edge, we can ignore it.

We will see that all possible partitionings of the nodes lead to a correct solution but most of them would not lead to the desired speed-up of the computation. If we consider a street network and hence a graph with a layout, intelligent partitions are those in which nodes of one partition are geometrically adjacent. We have examined several partitioning algorithms which are well applicable and give good speed improvements. Tested on real-world street networks we compared the different strategies with reference to the resulting average search space and to their preprocessing time.

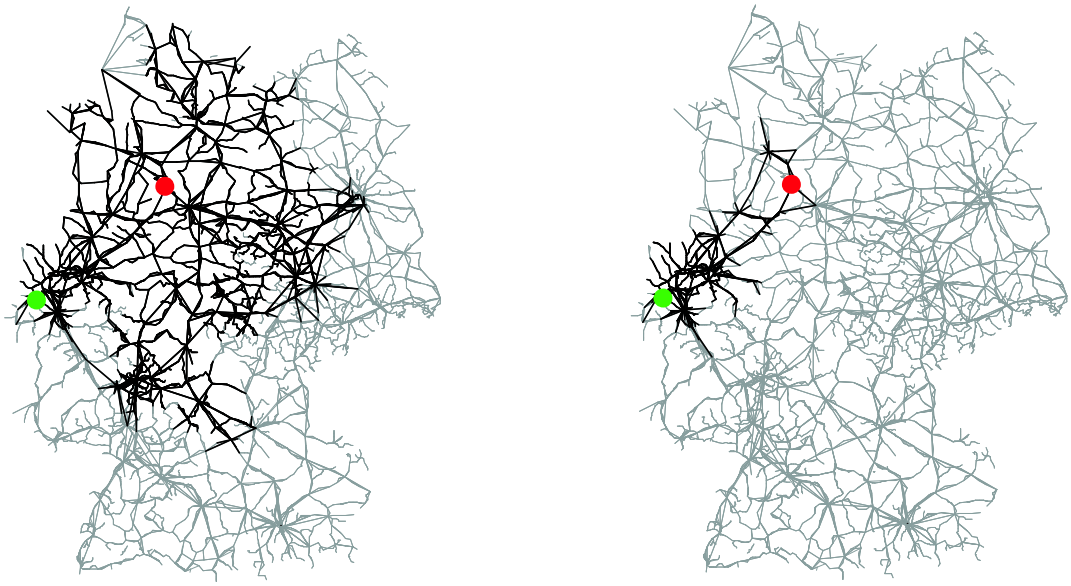


Figure 1: This figure illustrates the search space of a standard DIJKSTRA search (left) and an accelerated partition-based DIJKSTRA search (right). The source node is marked red, the target green, all touched edges during the search are drawn bold. The search stops, when the target node has been reached.

1.1 Definitions and Problem Description

The *single-source shortest-path problem* is probably one of the most studied problems in algorithmic graph theory. Consider a weighted graph $D = (V, E, l)$ where V is the set of nodes, $E \subset V^2$ the set of (directed) edges and l a function $l : E \longrightarrow \mathbb{R}$ which represents a weight for each edge (e.g. the Euclidian distance or the travel time of a train in a timetable-graph). Throughout the paper, we denote the number of nodes as n and the number of edges as m.

Let s and t be two nodes of the graph. The single-source shortest-path problem is to find the shortest path from the source node s to the target node t concerning the weight l – among all paths from s to t we search the one with the minimal weighted sum over all edges on the path. In general, there may be more than one shortest path from one node to another. However, if shortest paths are not unique, problems arise when different speed-up techniques are combining. We therefore assume uniqueness of the shortest paths, which can be achieved by adding a small fraction to the edge weights if necessary.

There are several algorithms to compute a shortest path between two nodes: *breadth-first-search* (if l is positive and constant), BELLMAN-FORD (no negative cycles) or the algorithm of DIJKSTRA which works on non-negative weights l[1].

[1]In the presence of negative weights but not negative cycles, it is possible, using Johnson's algorithm, to convert in $O(nm + n^2 \log n)$ time the graph to an equivalent one with non-negative edge weights that result in the same shortest paths [Joh77]

1.2 Dijkstra's Algorithm

We have already mentioned that we will use a modified version of DIJKSTRA'S ALGORITHM to calculate the shortest path between two nodes of a static graph D. The algorithm of DIJKSTRA can be considered as a weighted breadth-first search. We denote for each node $v \in V$ its current tentative distance from the start node as $d(v)$ and its predecessor node on the shortest path as $p(v)$. Furthermore, every node v gets a status $S(v)$, which can have the values {non-visited, labeled, visited}. In the initialization process, all nodes have the status non-visited, the distance $d(v) = \infty$ and the empty predecessor $p(v) = null$.

The nodes that are currently labeled by the algorithm are organized in a priority queue Q. At initialization, the source node s is marked as labeled with the distance $d(s) = 0$. While the priority queue is not empty, the algorithm extracts the node v from Q with the smallest current distance, scans it and sets its status to visited. "Scanning" of a node v means that all outgoing edges $(v, u) \in E$ are inspected and if the path $s - \cdots - v - u$ is shorter than $d(u)$, the distance $d(u)$ will be updated to $d(v) + l(u, v)$ and the predecessor is set to $p(u) = v$. If the node u is non-visited, it is inserted in the priority and the new status is labeled. If the node is already labeled, its priority in the priority queue is updated (i.e. decreased).

Following pseudo-code demonstrates the algorithm of DIJKSTRA, the status-flag can be omitted by testing, whether the node is a member of Q.

```
1  procedure DIJKSTRA(D = (V, E, l), s)
2    begin
3      d(s) := 0
4      Q.insert(s, 0)
5      while not Q.empty do begin
6        v := Q.extractMin
7        forall_out_edges(v, u) do begin
8          if d(u) ≤ d(v) + l(v, u) then
9            continue
10         d(u) = d(v) + l(v, u)
11         if u ∉ Q then
12           Q.insert(u)
13         else
14           Q.decreaseKey(u)
15       end
16     end
17   end
```

The worst-case time complexity of the algorithm depends on the type of the priority

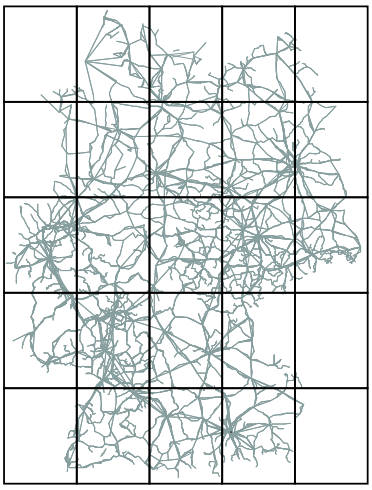


Figure 2: A 5×5 grid partition of Germany

queue. At least one of the operations *Insert* and *ExtractMin* must have $O(\log n)$ time complexity[2]. If we use, for example, a Fibonacci heap as priority queue, we get a worst-case time complexity for DIJKSTRA'S ALGORITHM of $O(m+n \log n)$ [CLRS]. There are also priority queues with lower time complexity bounds provided there are additional assumptions about the edge weight function (e.g. if l is integral and bounded [Gol01]).

The algorithm can be seen as a weighted breadth-first-search and hence if we consider a layout of the graph D and suppose that the weights of the edges are Euclidean distances, the search space of the algorithm spreads circularly around the source node s. Figure 1 illustrates this situation. The algorithm we will present, reduces this search space but it does not reduce the worst-case time complexity of DIJKSTRA'S ALGORITHM.

After a detailed description of the underlying algorithm in chapter 2 we will introduce several partitioning strategies of nodes in chapter 3. Chapter 4 provides an overview of the experimental results and chapter 5 will give some implementation details.

2 Partition-based Speed-Up

We want to speed-up the algorithm of DIJKSTRA by reducing the search space of each shortest-path search. The unmodified DIJKSTRA spreads circularly around the start node, and so it is possible that the algorithm, calculating a shortest path from Frankfurt to Hamburg in a road map of Germany, considers a shortest path via Munich—something a human would never do. Figure 1 illustrates this disaster. We will accomplish an impressive reduction of the search space by using preprocessed data during the search of our shortest-path.

[2]If not, we could sort an array of elements in less than $O(n \log n)$.

A sub-path of a shortest is also a shortest path, or more formally: if $s - n_1 - \cdots - n_k - t$ is a shortest path from s to t also $n_i - \cdots - n_k - t$ is a shortest-path from n_i to t. Suppose now, we would know for each edge e, if e is the beginning of a shortest path to our target t. With this information and the above property we can reduce our search space during the search with the following insight: if e is not the beginning of a shortest path to target t it can not be part of a shortest path from s to t.

We could now compute all shortest paths between all pairs of nodes and we would get, for each edge, the information if it is the beginning of a shortest path to a target t. But this method would need $\Theta(mn)$ space since we need this information for each edge and each node (or $\Theta(n^2)$ space if shortest paths are unique).

The idea is now the following: We do not calculate this information for each edge and all nodes but we partition the nodes of the graph into regions and determine if the edge e is the beginning of a shortest path to *any* node in a region. Using this information, we can ignore an edge e in a shortest path computation to a node t, provided our target node t is inside a region to which the edge e is not the beginning of a shortest path.

More precisely, we split the nodes of the graph in p regions with a function $r : V \longrightarrow \{1, \ldots, p\}$, which maps each node to a region number. For example, if we have a 2D-layout of the graph, we can lay a regular grid over this layout, enumerate the grid-cells and we get a partition of the nodes (see Figure 2). After the partitioning is defined by r, we introduce, for each edge of the graph, a *bit-vector* with p bits. The preprocessing of the algorithm marks in this bit-vector those regions to which e is part of a shortest-path. (For edge e, we set the bit i to `true` if e is part of a shortest path to a node in region i.) The space requirement for the preprocessed data is $\Theta(p \cdot m)$ for p regions because we have to store one bit for each region and edge. During the shortest-path search, while scanning a node n, the algorithm considers all outgoing edges and can now ignore those edges which have not set the bit for the region of the target node. The pseudo-code of the modified version of DIJKSTRA'S ALGORITHM could look like this:

```
1  procedure PartitionBasedDijkstra(D = (V, E, l), s, t)
2    begin
3      TargetRegion := r(t)
4      d(s) := 0
5      Q.insert(s, 0)
6      while not Q.empty do begin
7        v := Q.extractMin
8        forall_out_edges(v, u) do begin
9   →      if not Bitvector[(v, u), TargetRegion] then
10           continue
11         if d(u) ≤ d(v) + l(v, u) then
12           continue
13         d(u) = d(v) + l(v, u)
14         if u ∉ Q then
```

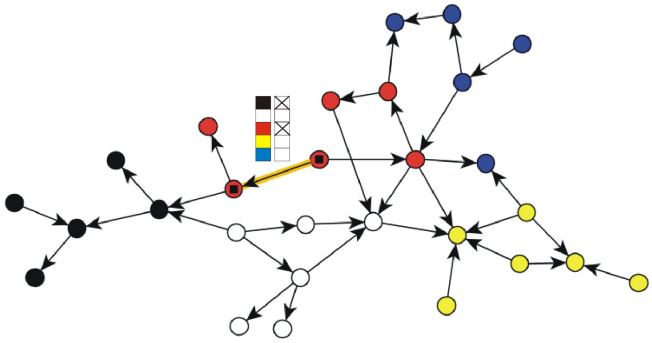


Figure 3: Illustration of the partition-based acceleration: the marked edge only leads to red and black nodes. A search with targets in blue, yellow or white regions can ignore this edge.

```
15              Q.insert(u)
16          else
17              Q.decreaseKey(u)
18          end
19      end
20  end
```

The main modification is in line 9. This modification does not effect the correctness but the running time of the search, which is proven in the following

Theorem. PartitionBasedDijkstra *finds the same shortest paths as* Dijkstra.

Proof. Let s and t be arbitrary but fixed nodes and P a shortest s-t-path that is found by Dijkstra. Since a sub-path of a shortest path is also a shortest path, each edge $e \in P$ is the beginning of a shortest path to the node t and hence the preprocessing has set the bit of the target-partition for the edge e. Therefore, PartitionBasedDijkstra uses this edge and finds this shortest-path. □

In the next section, we will have a look at the basis of the speed-up method—the preprocessing.

2.1 The Preprocessing

We have to calculate the bit-vectors for all edges. We can achieve this by calculating for every node n its shortest-path tree to all other nodes v (a simple Dijkstra search without target) and set the bit of partition $r(v)$ of the bit-vector of the first edge e of the shortest

path from n to v. After all nodes n have been considered, the bit-vectors of all edges are set correctly by this construction.

```
1  procedure PartitionBasedDijkstra-PrePocessing(D = (V, E, l))
2    begin
3      forall_edges(e) do
4        reset_bitvector(e)
5      forall_nodes(n) do begin
6        Dijkstra(D, n)
7        forall_nodes(v) do begin
8          e = first_edge_of_path(n, v)
9          set_bitvector(e, r(v))
10       end
11     end
12   end
```

The function *first_edge_of_path* can be calculated on-the-fly during the Dijkstra search. Setting the bit-vectors needs $O(mn)$ time for all pairs of edges and nodes. The running time for the preprocessing is therefore dominated by the time needed to compute n times Dijkstra, which has complexity $O(m + n \log n)$. The resulting time complexity is $O(n \cdot (m + n \log n + n))$. For sparse graphs $m = O(n)$ like typical traffic networks, we get a worst-case time complexity of $O(n^2 \log n)$. This is not satisfactory, because we want to have a scalable algorithm which means that it should run in linear time. Fortunately, we can give a scalable variant of the preprocessing.

2.2 Preprocessing without All-Pairs Shortest Paths

Fortunately, it is not necessary to compute all-pairs shortest paths to fill the bit-vectors correctly. We can do better using 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: if s is not a member of region R then there exists an edge $e = (u, v)$ with $r(u) \neq p_R$ and $r(v) = p_R$. We will see that it is sufficient if the preprocessing algorithm only regards shortest paths to such nodes v which are targets of *overlapping* edges. We will call such nodes *boundary-nodes*.

Theorem (Boundary-Nodes). *In the preprocessing, it is sufficient to consider only shortest paths to boundary-nodes (i.e. nodes v which are targets of edges $e = (n, v)$ with $r(u) \neq r(v)$.*

Proof. Let s and t be arbitrary but fixed nodes with the shortest path $s = n_1 - \cdots - n_k = t$ and different region-membership $p_s = r(s) \neq r(t) = p_t$. One can easily see by induction that there exists an edge $e = (n_i, n_{i+1})$ in this shortest path with $p_{n_i} = r(n_i) \neq r(n_{i+1}) = p_t$. The preprocess which only regards shortest paths to boundary-nodes would have regarded

the path from s to node n_{i+1} and hence it would have set the bit of region p_t on all edges of the shortest path $s - \cdots - n_{i+1}$ and the modified DIJKSTRA algorithm would find the shortest path from s to t (the region bits p_t of the edges between n_{i+1} and t are set, because these edges are members of region p_t and we statically set for all edges the bit for their corresponding region). □

But what does the algorithm do in order to only follow the shortest paths to boundary nodes? We define the *reverse-graph* D_{rev} of a directed graph $D = D(V, E, l)$ as the graph $D_{\text{rev}} = D(V, E_{\text{rev}}, l_{\text{rev}})$ with

$$E_{\text{rev}} = \{(u, v) | (v, u) \in E\} \quad \text{and} \quad l_{\text{rev}}(u, v) = l(v, u).$$

In words, the reverse-graph is the graph D with all edges reversed. It is easy to see that $s - \cdots - t$ is a shortest path from s to t in D iff $t - \cdots - s$ is a shortest path in D_{rev} with the same edges 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_{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 the shortest-path tree in D_{rev} and stores for each node v its predecessor edge e_v of the shortest path. In D_{rev} this is an incoming edge of v and hence in D it is an outgoing edge which is the beginning of the shortest path from v to b. For this edge e, the algorithm sets the region-bit of R in the bit-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 regarded and all region-bits of region R of all edges have been set.

This is an improvement of the first preprocess algorithm, because the algorithm does not calculate the shortest-path trees of *all* nodes of D—only the shortest-path trees of boundary nodes are calculated. We denote the number of boundary-nodes with k and get a time complexity of the improved preprocess of $O(k \cdot n \log n)$.

```
1  procedure PartitionBasedDijkstra-ImprovedPreProcess(D = (V, E, l))
2    begin
3      forall_edges(e) do
4        reset_bitvector(e)
5      forall_boundary_nodes(n) do begin
6        ReverseDijkstra(D, n)
7        forall_nodes(v) do begin
8          e = last_edge_of_path(n, v)
9          set_bitvector(e, r(n))
10       end
11     end
12   end
```

The number k of boundary nodes is highly dependent on the partitioning of the nodes. We can minimize k by taking the minimal edge-separator for a certain number of regions. However, we will see in section 4 that this partition does not always lead to the best results concerning the size of the search space. As an example for the resulting preprocessing time: This improved preprocess algorithm calculated the preprocessing data of one of our real-world graphs with 473000 nodes and $1,1$ million edges by using a 10×10 grid-partition in 2,5 hours and the average search space during the single requests was reduced to less than 4% compared to the standard search - this lead to a factor 34 improvement of time.

Another advantage of this speed-up method is the possibility of optimizing it for important nodes of the graph. Consider a demand analysis of the customer queries with the result that a small set of nodes is often the target of most of the requests (e.g., the railway station of Berlin or the airport of Frankfurt in the transport network of Germany). It is a common approach to store the shortest paths to these most important nodes. However, the same effect can be achieved within the framework of the partition-based speed-up technique: If each of these important nodes is assigned to its own, specific region, the modified shortest-path algorithm will find the direct path without regarding unnecessary edges or nodes. Storing the shortest paths to important nodes can therefore be realized without any additional implementation effort.

It is easy to see that the preprocessing algorithm can easily be adapted to a parallel algorithm—each processor calculates the shortest-path trees of a subset of the boundary-nodes B_R. The results are independent.

2.3 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. Consider the shortest-path tree of a node v_1 and a node v_2 in the same region as v_1. For every node $v \in V$ of the graph, we get an upper bound of the length of the shortest path from v_2 to v: it can not be longer than the distance from v_2 to v_1 and the shortest path from v_1 to v. We can now use this upper bound to reduce our preprocessing effort.

For the first boundary-node v_1 of each region, the algorithm computes its (reverse) shortest-path tree, the corresponding bit-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 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 bit-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.

Experiments show, that this method reduces the number of nodes that are put into the priority queue during the preprocess to less than 70%. The running time was improved by up to 20 %.

Table 1: Analysis of the bit-vectors: kdTree(n) and METIS(n) are partitioning algorithms of size n (compare with chapter 4). For 80% of the edges, either almost none or nearly all bits of the corresponding bit-vector are set.

Graph	#Edges	Algorithm	$= 1$	$< 10\ \%$	$> 95\ \%$
street_network_1	920,000	KdTree(32)	351,255	443,600	312,021
street_network_1	920,000	KdTree(64)	334,533	470,818	294,664
street_network_1	920,000	METIS(80)	346,935	468,101	290,332
street_network_4	2,534,000	KdTree(32)	960,779	1,171,877	854,670
street_network_4	2,534,000	KdTree(64)	913,605	1,209,353	799,206

2.4 Reduction of the Preprocessed Data

An analysis of the calculated bit-vectors shows that there exists a possibility for (lossy) compression of the bit-vectors: For 80% of the edges either almost none or nearly all bits of their bit-vectors are set. Table 1 shows an excerpt of the analysis we made.
The column "= 1" shows the number of edges, which are only responsible for shortest-paths inside their own region (only one bit is set). Edges with more than 95% bits set, could be important roads. Of course, these results are highly dependent on the characteristics of the graphs, but probably this can be recognized for other street-networks, too.

This justifies ideas for (lossy) compression of the bit-vectors, but it is important that the decompression algorithm is very fast—otherwise the speed-up of time will be lost. The two-level technique, described in the following chapter, is a suitable lossy compression method for the bit-vector entries.

2.4.1 Two-Level Partitions

As illustrated in figure 1, the modified Dijkstra search reduces the search space next to the beginning of the search but once the target region has been reached, all nodes and edges are visited. This is not very surprising if you consider that all edges of a region have set the region-bit of their own region. We could handle this problem if we used a finer partitioning of the graph but this would lead to longer bit-vectors (requiring more memory). As an 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 edge. However, the information for the fine grid is mainly needed for edges next to the target node. This leads to the idea that we could split each region of the coarse partition but store this additional data only for the edges inside the same coarse region. More precisely, we can look at the target region as if it was an independent graph. Then, we can partition this graph again and perform an additional preprocessing with bit-vectors on each edge. Therefore, each edge 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

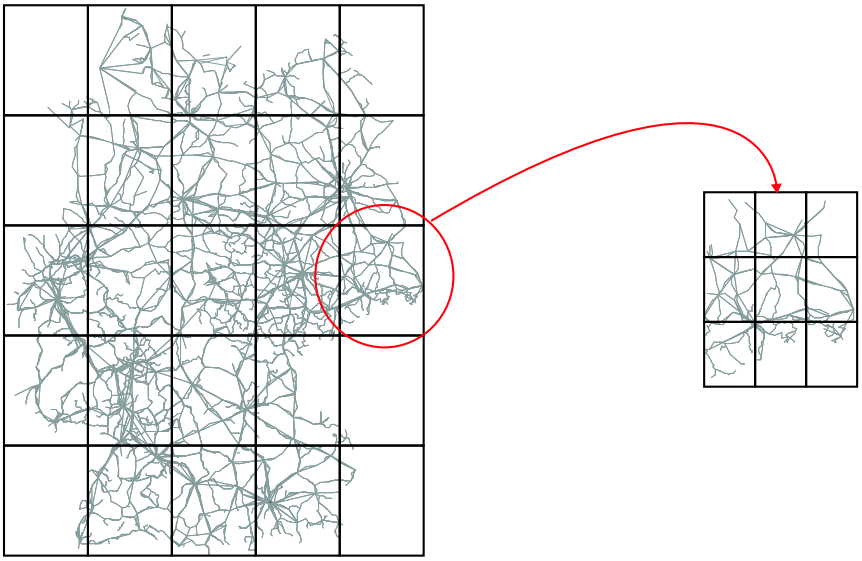


Figure 4: Illustration of a two-level partition. The first-level is a 5×5 grid and each coarse region is partitioned in the second-level by a 3×3 grid. For each edge, the bit-vector for the coarse grid and the bit-vector for the fine grid *in the same region as the edge* is stored.

one-level partitioning, because the second bit-vector exists only for the target region (34 bits per edge 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 bit-vectors of neighbored regions are similar for regions far away. Thus, we could see this two-level method as a compression of the first-level bit-vectors: we summarize the bits for remote regions: if one bit is set for one, the bit is set for the whole group; a kind of lossy compression.

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

```
1  procedure PartitionBasedTwoLevelDijkstra(D = (V, E, l), s, t)
2    begin
3      TargetRegion := r(t)
4      SubTargetRegion = st(t)
5      d(s) := 0
6      Q.insert(s, 0)
7      while not Q.empty do begin
8        v := Q.extractMin
9        forall_out_edges(v, u) do begin
10   →       if not BitvectorFirstLevel[(v, u), TargetRegion] then
11             continue
12   →       if (v, u) ∈ TargetRegion then
13   →         if not BitvectorSecondLevel[(v, u), SubTargetRegion] then
14             continue
15           if d(u) ≤ d(v) + l(v, u) then
16             continue
17           d(u) = d(v) + l(v, u)
18           if u ∉ Q then
19             Q.insert(u)
20           else
21             Q.decreaseKey(u)
22       end
23     end
24   end
```

Experiments showed (compare with chapter 4) that this method leads to the best results concerning the reduction of the search space - but an increased preprocessing effort is required.

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 marked.

2.5 Bidirectional Search

The bidirectional search leads to speed-up factors of about 4 in the unaccelerated case. (Figure 5 illustrates the search space in contrast to the standard Dijkstra). Instead of performing one search from the source node, a second, backward search from the target node is started in parallel. The algorithm can stop if both search horizons meet, or more precisely 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. (Note that v is not necessarily on that shortest path.) In order to avoid a search for the connector-node of the two searches, we determine the shortest path on-the-fly: every time we consider a

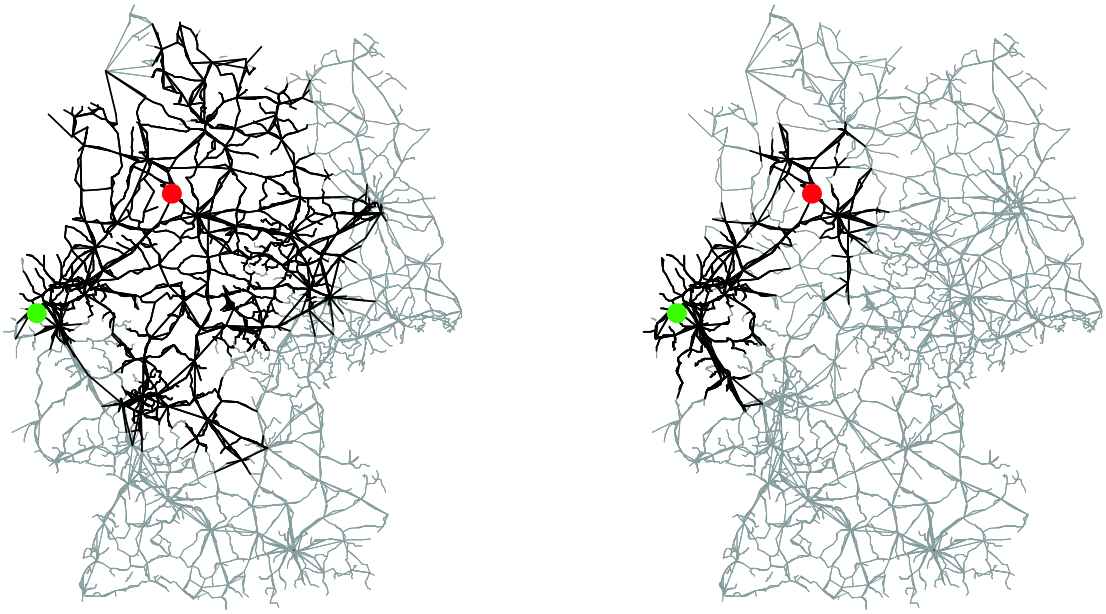


Figure 5: Reduction of the search space by using a bidirectional search (the search space of a standard DIJKSTRA search is shown on the left. The right figure illustrates the search space of a bidirectional search.).

node which is labeled by both directions, we update the minimal sum of the shortest paths to source and target.

In principle, this speed-up method can be combined with any other one. The best results in our experiments (with fixed total number of bits per edge) 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). The preprocessing for both directions must be computed independently. The reverse graph R_D is used for the calculation of the bit-vectors for the backward search.

3 Partitioning Algorithms

In the last section, we described a speed-up technique that uses a partitioning of the graph to precompute information whether an edge may be part of a shortest path. Any possible partitioning can be used and the algorithm would return a shortest path. However, most partitionings do not lead to an acceleration if used for the partition-based speed-up. In this section, we will present the partitioning algorithms that we used to reduce the search space of DIJKSTRA'S ALGORITHM. Most of these algorithms need a layout of the graph. In our case, this is 2D layout, but all partitioning algorithms can be adapted easily to higher dimensions.

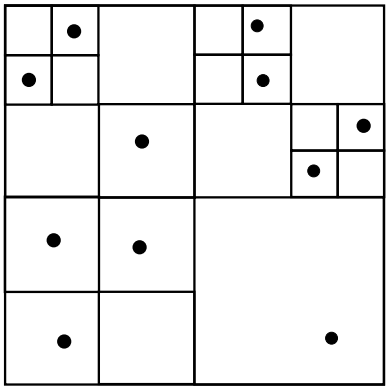


Figure 6: Example of a QuadTree.

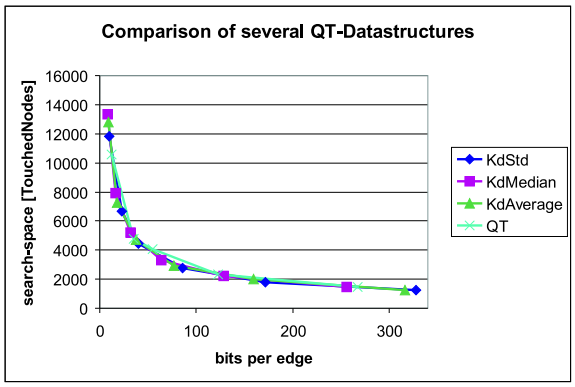


Figure 7: Comparison of several kd-tree partitions. The difference of the resulting search space is marginal.

3.1 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 $(l, t)^T$ the top-left coordinate of the bounding box of the 2D layout of the graph and with $(r, b)^T$ the bottom-right one. Furthermore, we define $w = r - l$ as the width and $h = t - b$ as the height of the layout. The grid cell $G_{i,j}$ with $0 \leq i < n, 0 \leq j < m$ is now defined as the rectangle

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

(Nodes on a grid line are assigned to an arbitrary but fixed grid cell.) Figure 2 shows an example of a 5×5 grid.

This 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 grid partitioning has always the worst results in our experiments.) Because earlier works on this speed-up method used this partitioning method, we use the grid partitioning as a baseline and compared all other partitioning algorithms with it.

3.2 Quadtrees

A *quadtree* is a data structure for storing points in the plain. Especially in algorithmic geometry this data structure is used, because nearest neighbors can be found very fast. Quadtrees can be generalized to higher dimensions—for 3D they is called *octrees*.

Definition (QuadTree). *Let P be a set of n points in the plain, R_0 its quadratic bounding-box, then the data structure* quadtree *is recursively defined as follows:*

- *Root v_0 corresponds to the bounding-region R_0*

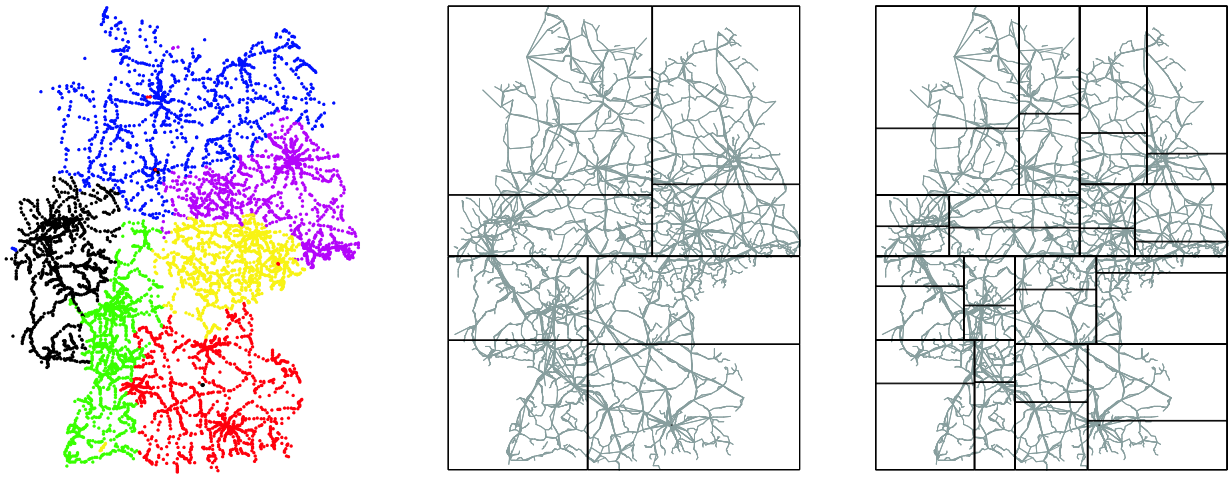


Figure 8: Visualizations of partitions of Germany: 6 regions with METIS (left), 8 regions with a kd-tree (middle) and 32 regions with kd-tree (right).

- *R_0 and all other regions R_i are recursively divided into four quadratic sub-regions, until each non-empty region contains exactly one point of P. The four subregions of R_i are sub-nodes of v_i in the tree.*

Because, for our application, we do not want to create a separate region for each node, we modify this definition. 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 least one node and at most b nodes. 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 divided into four equally-sized sub-regions. This division can be extended to more general sub-division schemes. This leads to the definition of a *kd-tree*. In the construction of a *kd-tree*, the plane is recursively divided similar as for a quadtree. However, the underlying rectangle is decomposed into *two halves* by a straight line parallel to an axis. The partition axes are alternate in the order $x, y, x, y, \ldots$. The positions of the dividing line may depend on the data. Frequently used positions are given by the center of the rectangle or the *median* of the points. In applications with higher dimensions, the partition axes are not cycled but the dimension with the largest variance is used.

Experiments on our real street networks showed that the *kd-tree* with median partition position usually leads to the best results (figure 7). Therefore, we only used this method as a representative for this partitioning class. If the median of the points is used, the partitioning has always 2^l regions and 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 transfered 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 [CLRS]. 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 175s, the bit-vectors were calculated in seven hours.)

3.4 METIS

METIS is a software package for partitioning graphs into k equally-sized parts with regard to optimize the number of edges that separates partitions to a minimum [Kar95]. It can be obtained free-of-charge for all common platforms. There are two advantages of this method for our application. The METIS algorithm does not need a layout of the graph and the preprocessing is faster because the number of separating edges is noticeable smaller than in the other partitioning methods. Figure 8 shows a partitioning of a graph generated by METIS.

4 Experimental Results

The main goal of this work is to compare the different partitioning algorithms concerning their resulting search space and speed-up of time during the accelerated DIJKSTRA search. We tested eleven combinations of the above described techniques. 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

The bidirectional search can be accelerated by our partition-based method in both directions. The partitioning can differ for the two directions. Since computing all possible combinations on all graphs takes way to much time, table 2 shows the selection of tested algorithms.

We tested the algorithms on German street-networks, which are directed and have a 2D layout and positive, integral edge weights. Table 3 shows some characteristics of the graphs. The Shortest-Path column is the average number of nodes on a shortest path in the graph, *TargetDijkstra* is a standard DIJKSTRA search, which stops if the target node is reached.

Table 2: 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 kd-tree 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

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.

Abstractly, 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 are interested in two results: What is the *size of the reduced graph* for the chosen partitioning method and what is the resulting *speed-up of time*. The first is an indication of the quality of the algorithm and the latter is an indication, if this speed-up method is practicable and reasonable in the application. We used the number of Touched Nodes (the number of nodes, which are put into the priority queue during the search) as measurement for the search space. Fortunately, the additional test during the DIJKSTRA search is only to test a bit of a bit vector and this does not lead to a significant overhead. The TargetDijkstra is used as a reference algorithm to compare the number of Touched Nodes and the speed-up of time.

Most of the shown figures compare the speed-up of time or the reduction of the search space in relation to the size of preprocessed data which is the size of the calculated bit vectors.

Table 3: Overview of the tested street networks

Graph	#nodes	#edges	Shortest Path	TargetDijkstra [time]	TargetDijkstra [#TouchedNodes]
street_network_1	362,000	920,000	250	0,26s	183,509
street_network_2	474,000	1,169,000	440	0,27s	240,421
street_network_3	609,000	1,534,000	580	0,3s	306,607
street_network_4	1,046,000	2,534,000	490	0,78s	522,850

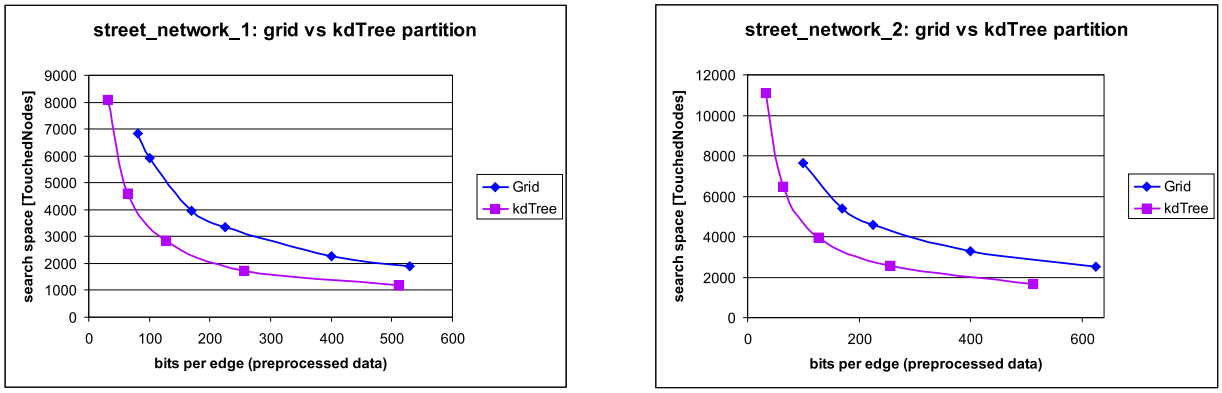


Figure 9: Comparison of the quality between grid partitions and *kd*-tree partitionings for unidirectional shortest-path searches. The usage of kd-tree partitions leads to clearly better results because it adapts to the layout of the graph.

4.1 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 tested graphs, the grid partitioning leads to smaller reductions of the search space, because the *kd*-tree adapts better to the 2D layout of the graph. Therefore, we will restict the further analysis of the partition-based DIJKSTRA to the *kd*-tree partitionings.

Figure 10 illustrates that, in our tested graphs, there is a linear correlation between the search space and the CPU time. This justifies that in the further analysis it is sufficient to consider the search space only.

Of course, the size of the absolute search space depends of the size of the graph. A surprising result is, that the relative size of the search space (relative to the search space of the TargetDijkstra) is very similar for all graphs (except street_network_3 which seems to have a special characteristic). The speed-up diagram of figure 11 shows the relative average time of one shortest-path search with respect to the time of the TargetDijkstra. (A single accelerated unidirectional search in network_1 with 128 bit preprocessed data per edge takes less than 3ms).

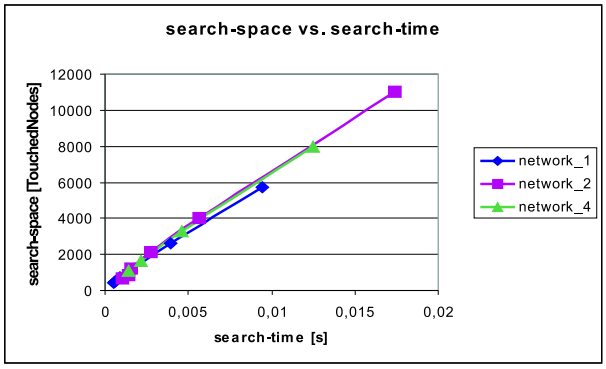


Figure 10: search space vs. search-time of bidirected, accelerated searches. Because of the linearity, it is sufficient to compare the search spaces in further analysis.

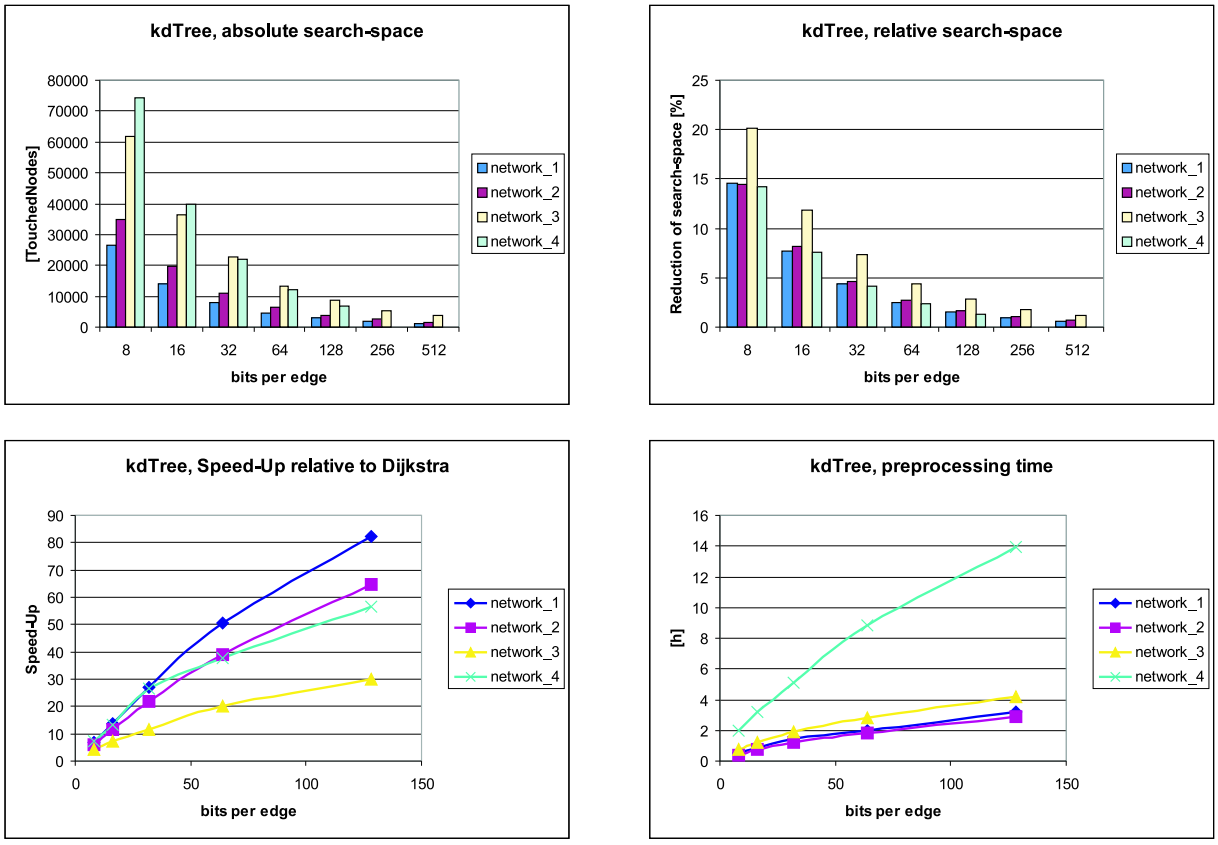


Figure 11: Results of our unidirectional algorithm using kd-tree partitions.

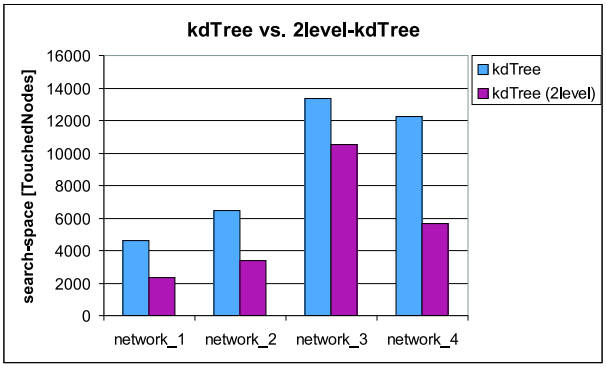


Figure 12: Comparison of one-level *kd*-tree (64 regions) and two-level kd-tree (64 first-level regions, 8 second-level regions).

4.1.1 Two-Level Partitionings

There was one reason for the introduction of the second-level partitions: if the shortest-path search enters the region of the target node t, all edges in the target regions have to be regarded by the algorithm. The second-level bit-vectors reduce the shortest-path search on the home stretch. Figure 12 compares the search spaces of the one-level and two-level accelerated searches - a reduction of factor 2 could be realised.

4.2 Bidirectional Search

The best results were achieved by a bidirectional DIJKSTRA search, which is accelerated in both directions: We can measure speed-ups that are more than 600 times faster than the TargetDijkstra algorithm. In general, the speed-up increases with the size of the graph.

Using a bidirectional search, the two-level strategy becomes less important, because the second-level bit-vectors will not be used in most of the shortest-path searches: The second-level bit-vectors are only used, if the search enters the region of the target. During a bidirectional search, the probabililty is high that the two search horizons meet in a different region than the source or target region. Therefore, the second-level bit-vectors are only used if both nodes are lying in the same region. Figure 13 confirms this estimation: only for large partitions in the first level is a speed-up recognizable with two-level bit-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 14 shows the results of the bidirectional search which is accelerated by kd-tree partitionings. Even with less preprocessed data (16 bits per edge), we get a speed-up of over 50. The accelerated search on network_4 is 545 times faster than the TargetDijkstra, when using 128 bits per edge preprocessed data (1,3 ms per search).

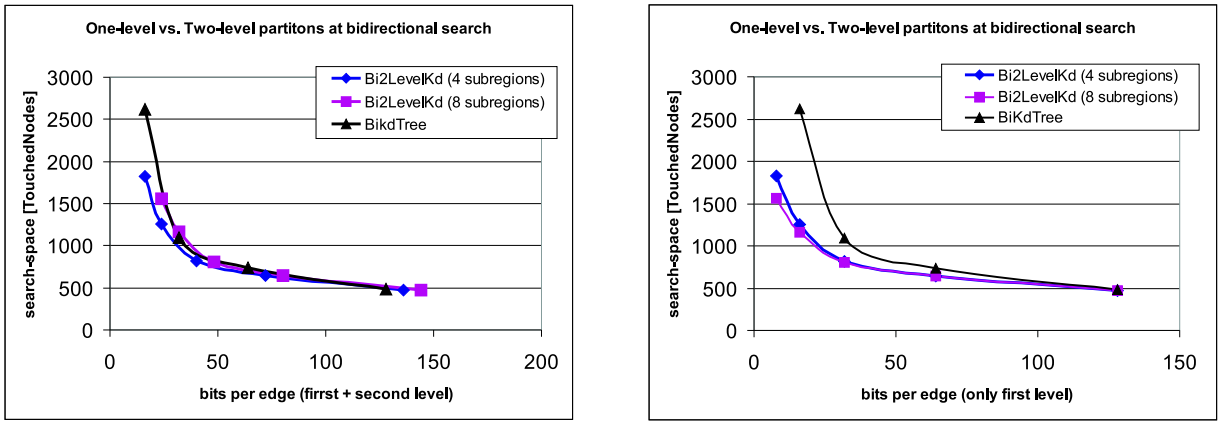


Figure 13: The two-level strategy becomes irrelevant for the bidirectional search case. If more than 32 regions for the first-level are used, the two-level acceleration provides no noticable improvement.

4.3 Comparison of the Partitioning Methods

Figure 15 compares the results of our several partitioning methods on the four street networks. The size of the preprocessed data is nearly the same for all algorithms. (The same size could not be realized as, e.g., kd-tree partitionings always have size 2^l.) Table 4 shows the used partitionings for the comparison.

For the unidirectional searches, the two-level strategies lead to 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 are marginal between the partitioning techniques.

In summary, the best of our tested partition-based speed-up methods is a bidirected search, accelerated in both directions with kd-tree partitions or METIS partitions.

5 Implementation

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 ([MN99]).

In order to measure the unaffected improvement in performance with respect to time, we implemented the algorithms very carefully without using frameworks for re-use of code. Experiments showed, that using complex class-hierarchies with virtual functions leads to an aggravation of 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 edge data to store the predecessor edge or the current distance of this node. To get comparable results, we are computing several shortest-path requests (the requests are listed in the demand-file) - at a

Table 4: Used partitions for the comparison with nearly the same preprocessed data size

Name of partitioning	forward		backward		bits per edge
	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

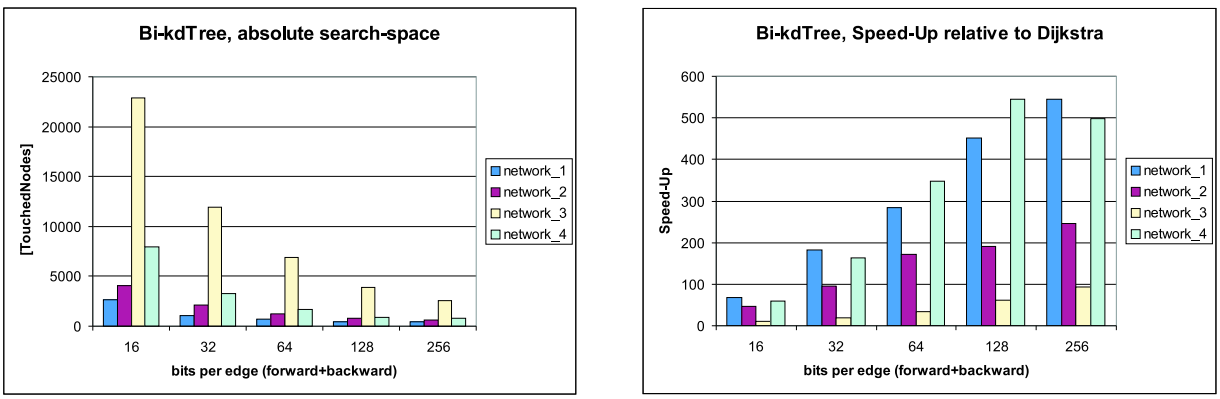


Figure 14: search space and speed-up of our algorithm with a bidirected, accelerated search using kd-tree partitions.

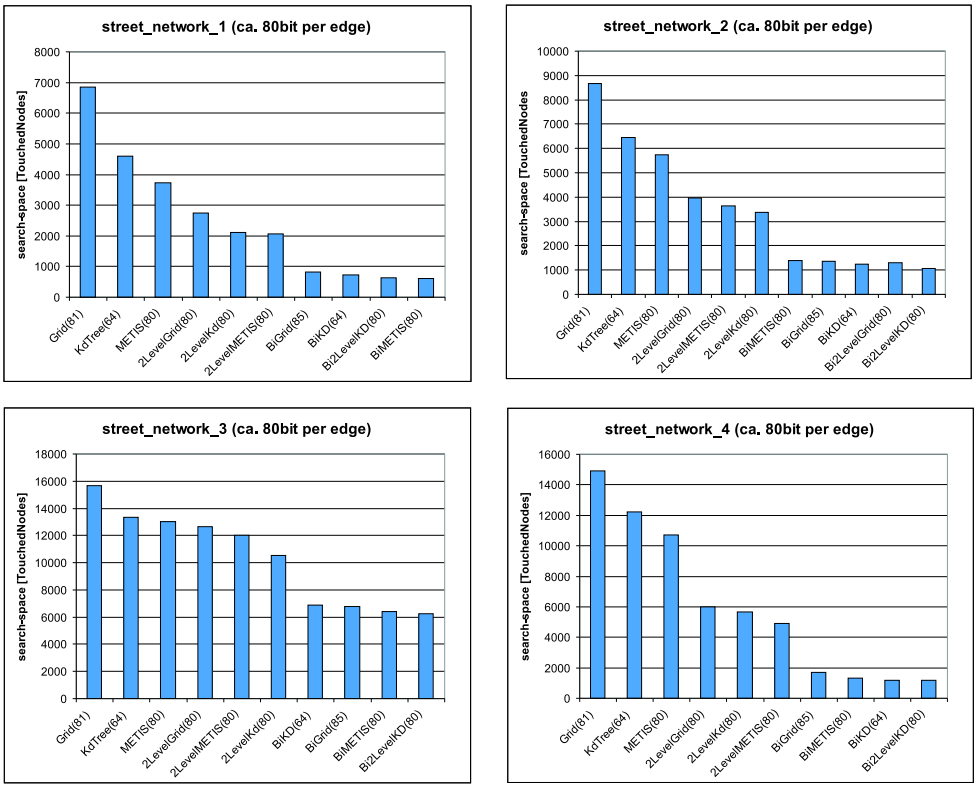


Figure 15: Comparison of most of the implemented algorithms on the four street networks. The parameter of the used partitions are listed in the table 4. The number of bits are noted in brackets.

minimum of 1000 requests. Because an initialization at the beginning of each shortest-path search would take a long time, we used time-stamps: every time, a node is touched by the search it gets the time of the current search and hence we know, if the data of the arrays is valid for this search or not - the initialization step of all nodes can be omitted. This is also a suitable method in the real application - where a central server has to answer a huge number of shortest-path queries.

For the bidirectional search there are several possible methods of choosing the current calculating direction. In order to reduce the resulting search-space, our algorithm takes that direction with the smaller search-horizon (the size of the priority queue).

6 Conclusion and Outlook

We have seen that it is possible to reduce the time of a single shortest-path search in a static street-network up to a factor of 500 with the presented speed-up technique. The preprocessing effort scales well and can easily be adapted to a parallel algorithm: the bit-vector entries of different regions are independent and can be computed in parallel.

Of the tested partitioning methods, we can recommend the kd-tree used for forward- and backward acceleration, because it leads—besides the METIS partitioning—to the best results and is easy to implement.

It would be interesting to find optimal partitions for this speed-up technique. One approach for finding nearly optimal partitions could be found in a kind of clustering method: Consider the bit-vectors of the preprocessing of the partition with exactly one node per region and regard the Hamming-distance according to the bit-vectors of two nodes (the number of bits that differ for these two nodes (single-node regions) over all edges). These distances form a distance matrix. We can now search a partitioning with k regions where the distances of nodes in one region are minimal according to the distance matrix. This problem is known as *clustering*. There are several algorithms for clustering data with a distance matrix ([Wag03] provides an overview).

The problem with this method is that the creation of the distance matrix for the nodes has $O(n^3)$ time complexity. For each pair of nodes we have to compare the bits of all edges. By new means, it may be possible to reduce this time complexity. Since this is clearly beyond the scope of this paper, no experiments and implementations were made.

References

[CLRS] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. The MIT Press.

[Gol01] Andrew V. Goldberg. A simple shortest path algorithm with linear average time. In *proceedings of the 9th European Symposium on Algorithms (ESA '01), Springer Lecture Notes in Computer Science LNCS 2161*, pages 230–241, 2001.

[Joh77] D. B. Johnson. Effcient algorithms for shortest paths in sparse networks. *Journal of the ACM (JACM)*, page 24(1):1, 1977.

[Kar95] George Karypis. METIS: Family of multilevel partitioning algorithms. `http://www-users.cs.umn.edu/~karypis/metis/`, 1995.

[MN99] K. Mehlhorn and S. Näher. LEDA, a platform for combinatorial and geometric computing. *Cambridge University Press*, 1999.

[SWW00] Frank Schulz, Dorothea Wagner, and Karsten Weihe. Dijkstra's algorithm online: An empirical case study from public railroad transport. *Journal of Experimental Algorithmics*, 5(12), 2000.

[SWZ02] Frank Schulz, Dorothea Wagner, and Christos Zaroliagis. Using multi-level graphs for timetable information in railway systems. In *Proceedings 4th Workshop on Algorithm Engineering and Experiments (ALENEX)*, volume 2409 of *LNCS*, pages 43–59. Springer, 2002.

[Wag03] Silke Wagner. Optimierung des Demographic Clustering Algorithmus. Master's thesis, Dept. of Informatics, University of Konstanz, Germany, February 2003.

[WWZ04] Dorothea Wagner, Thomas Willhalm, and Christos Zaroliagis. Geometric shortest path containers. Technical Report 2004-5, Universität Karlsruhe, Fakultät für Informatik, 2004.