Goal Directed Shortest Path Queries Using **P**recomputed **C**luster **D**istances*†

Jens Maue
Inst. of Theoretical Computer Science, ETH Zürich, Switzerland
jens.maue@inf.ethz.ch

Peter Sanders
Fakultät für Informatik, Universität Karlsruhe (TH), Germany
sanders@ira.uka.de

Domagoj Matijevic
Dep. of Mathematics, J.J. Strossmayer University, Osijek, Croatia
domagoj@mathos.hr

Abstract

We demonstrate how Dijkstra's algorithm for shortest path queries can be accelerated by using precomputed shortest path distances. Our approach allows a completely flexible tradeoff between query time and space consumption for precomputed distances. In particular, sublinear space is sufficient to give the search a strong "sense of direction". We evaluate our approach experimentally using large, real-world road networks.

Keywords: graph decomposition, preprocessing heuristics, road networks, shortest paths

1 Introduction

Computing the shortest path between two points in a network is one of the most fundamental algorithmic problems. There are many real-world applications that translate to this problem, one of which is answering optimal-path queries in route planning systems such as timetable information services for public transport or car navigation systems.

A well studied and widely used algorithm for shortest paths problems is Dijkstra's algorithm [4]. The asymptotic running time of Dijkstra's algorithm is $O(m + n \log n)$, where n is the number of nodes, and m is the number of edges. This reduces to $O(n \log n)$ for sparse graphs. The road networks considered in this paper are almost planar and thus highly sparse. Though Dijkstra's

*Partially supported by DFG grants SA 933/1-2,1-3. Part of this work was done at Max-Planck-Institut für Informatik, Saarbrücken, Germany.

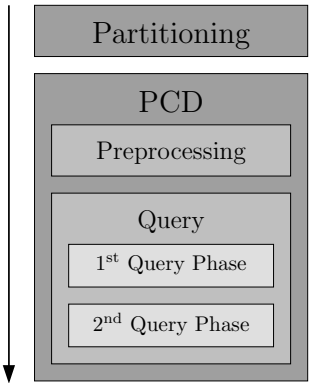


Figure 1: Components of PCD featuring the preprocessing and the query part. The preceding partitioning algorithm can be chosen independently of PCD.

algorithm is in fact optimal in a strong theoretical sense, it is often too slow for answering single-source single-target queries, particularly for applications to large graphs with frequent queries.

In the typical application scenarios, the queries are to be answered exactly and quickly, and there are usually many queries while the network does not change. This is a motivation for performing some amount of preprocessing in order to improve the query times. Naturally, precomputing and storing the shortest path distance for *all* possible queries would result in constant time queries—if only the distance is required as the answer. But, this approach is prohibited by its huge time requirement, and the quadratic space is not affordable either, which holds for large graphs in particular. Therefore, the preprocessing has to work reasonably fast, and the amount of preprocessed data should not exceed that of the input. On the other hand, the queries ought to be effectively accelerated independently of the graph size. These aspects present conflicting objectives, and to allow adjusting the trade-off between them is a desirable feature of a possible algorithm.

To sum up, this article focuses on *exact* single-source single-target shortest path queries on large street networks. The main goals are *fast queries* on a temporarily static graph without a layout or an embedding (e.g. without any knowledge of node coordinates), using a *fast preprocessing* that stores a *small* and *adjustable amount of auxiliary data*.

Precomputed Cluster Distances (PCD)

The PCD algorithm presents a goal-directed speedup heuristic with preprocessing based on Dijkstra's algorithm. (Cf. the classification of shortest path heuristics in Figure 2.) As outlined in Figure 1, PCD comprises two parts, a preprocessing and a query algorithm.

As its input, the preprocessing algorithm expects a weighted graph which has been partitioned into k clusters. (There are no further assumptions about this clustering, and PCD actually works correctly with any partitioning. As we will see later, the performance of PCD, however, depends on the chosen partitioning method.) For every pair of clusters, the preprocessing now computes

the minimum distance between any pair of nodes with one node from either cluster. This distance is stored for all k^2 pairs of clusters, which finishes the preprocessing.

Second, during a query, these precomputed cluster distances are applied in the following way: basically, Dijkstra's algorithm is executed (see Sect. 2.1) combined with bidirectionsl search (see Sect. 2.2). During the execution, we maintain an upper bound on the length of the shortest path we are searching, which is calculated repeatedly using the precomputed distances as outlined in the next paragraph. Additionally, when the query algorithm considers a node, the precomputed cluster distances provide a lower bound for the remaining distance from a node to the query target, from which a lower bound is calculated on the length of the shortest path we are looking for. If this lower estimate exceeds the current upper bound, this node cannot be a member of the shortest path and is therefore pruned.

The upper bound always represents the length of an actual path—not necessarily the shortest—from the query source to the query target. Whenever the algorithm reaches the start node of a precomputed shortest path to the cluster containing the target, it tries to tighten the upper bound. A lower bound is estimated in every step of the algorithm, and their quality depends on the number of clusters k.

Outline

The remainder of this paper is structured as follows. Section 2 gives an overview over shortest paths research and concepts related to PCD. Section 3 introduces the definitions and notation used in the rest of the paper. The next two sections contain the core idea of the paper, with Section 4 explaining the preprocessing algorithm, which constitutes the first part of PCD, and Section 5 containing a detailed description of the PCD query algorithm with its two consecutive phases. We then present several graph partitioning algorithms in Section 6 used to provide the preprocessing with some clustering. Experimental evaluation of PCD using large real-world graph instances is presented in Section 7. Query performance, space requirement, and preprocessing time are examined, and how these results are affected by varying the graph partitioning method, by different edge weight functions, and by the amount of preprocessing in particular. Section 8 summarises the main results and outlines possible future work related to PCD. This includes different graph partitioning algorithms, improvements of the query algorithm itself, and combining it with other speedup heuristics.

2 Related Work

A huge amount of research concerning shortest paths and related problems has been done in the recent past, and the main results related to PCD are presented in the following. For a recent overview refer to [3]. This particularly includes speedup heuristics based on Dijkstra's algorithm; Figure 2 gives an overview of the presented techniques, most of which perform some precomputation on the input data to speed up the search as motivated above. All preprocessing methods present different solutions to the problem of balancing between preprocessing time, additional space, and query performance. The latter can be rated

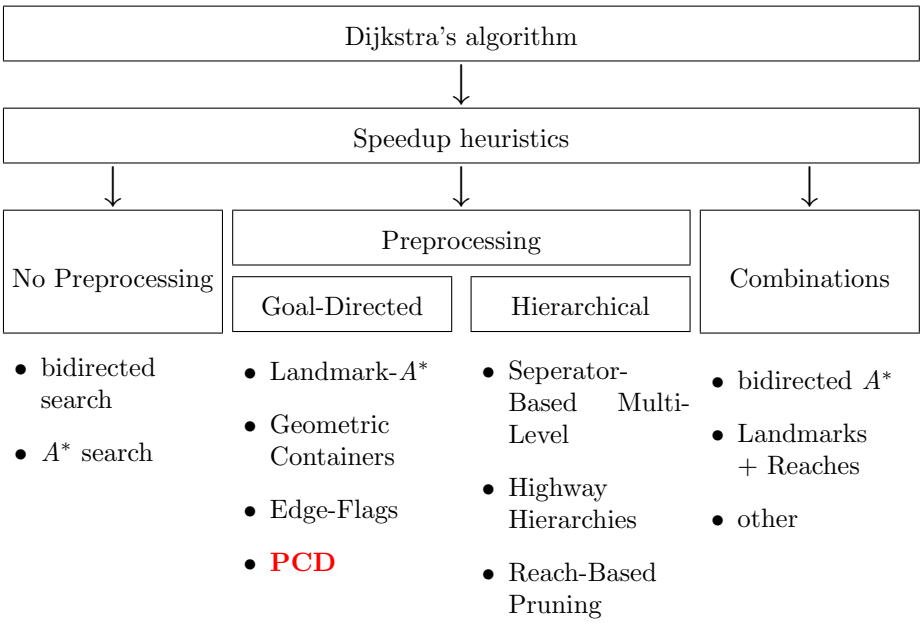


Figure 2: Overview of the shortest path algorithms and speedup heuristics introduced in this section.

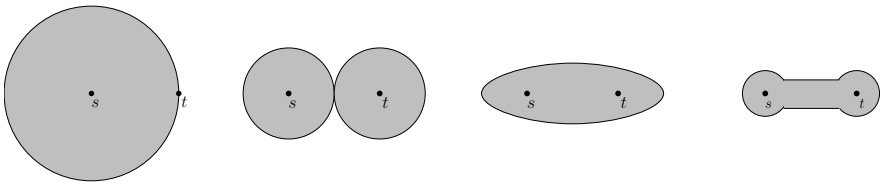
by a method's *speedup*, which denotes the factor by which an average query outperforms a search with Dijkstra's algorithm in terms of query time or size of the search space.

2.1 Dijkstra's Algorithm

The best-known and most commonly used shortest path algorithm is that of Dijkstra [4], which solves the single-source shortest paths problem for directed graphs with non-negative edge weights. Dijkstra's algorithm is efficiently implemented by using a priority queue, on which at most n insertions, n deletions, and up to m decrease-key operations are performed. Every node is inserted at most once, and its key is decreased at most once for every incoming edge until it is removed from the queue. A node that has been removed is also called a *settled* node as it will not be inserted again. The actual running time clearly depends on how this priority queue is implemented. The Fibonacci heap data structure allows insert and decrease-key operations in constant and deleting in $\mathcal{O}(\log n)$ amortised time, which yields a running time of $\mathcal{O}(m + n \log n)$ [5].

Though optimal for general ordered sets supporting only comparisons, the running time of $\mathcal{O}(m + n \log n)$ can be improved on for standard word RAM modelling [10]. A worst case time of $\mathcal{O}(m+n \log \log n)$ [27] is the best currently-known bound. Moreover, the single-source shortest paths problem can be solved in linear time for restricted families of graphs, such as undirected graphs [26], planar graphs [12], or uniformly distributed edge weights [22, 7].

Dijkstra's algorithm also solves single-source single-target shortest path queries, in which the search can be stopped when the distance to the target has been obtained. Though not improving the worst-case running time, there



(a) Dijkstra (b) Bidirectional (c) A^* Search (d) PCD

Figure 3: Idealized shape of the search space for a query from s to t explored by Dijkstra's algorithm, its bidirectional version, A^*, and PCD.

are many techniques that heuristically speed up such queries preserving correct solutions.

2.2 Search Heuristics

A quite simple acceleration technique not requiring any additional information such as a graph layout or preprocessed data is *bidirectional search* [24]. This simultaneously explores the reverse graph from the target and finishes when the search frontiers meet as sketched in Figure 3. As a rule of thumb, the search space of Dijkstra's algorithm is usually considered to grow by the square of the path distance for road networks, so bidirectional search can be expected to yield a speedup of two: if Dijkstra's algorithm explores p^2 nodes for a path of length p, its bidirectional version searches $2 \cdot \left(\frac{p}{2}\right)^2 = \frac{p^2}{2}$ nodes, i.e. half the number. The PCD query algorithm also applies bidirectional search.

The goal-directed A^* *algorithm* [11] reduces the search space by preferring nodes which are on a path with a low length estimate: given a potential function $\pi : V \to \mathbb{R}$, A^* repeatedly selects the node u whose estimate $d(s,u) + \pi(u)$ is the smallest; the resulting search space is shown in Figure 3. A^* corresponds to Dijkstra's algorithm in the following way: the *reduced weight* $w_\pi(e)$ of an edge $e = (u,v)$ is defined by $w_\pi(e) = w(e) - \pi(u) + \pi(v)$, and a potential function is called *consistent* if the reduced weight function is non-negative. For a consistent potential function, running the A^* algorithm is equivalent to running Dijkstra's algorithm on the graph with reduced edge weights, whereas Dijkstra's algorithm can be regarded as A^* with the zero-potential function. Euclidean distances provide a consistent potential function, but require an input graph provided with node coordinates and perform not very well for quickest path queries (e.g. weight of an edge is Euclidean distance it covers divided by its travel time). Unfortunately, the cluster distances precomputed by PCD do not provide a consistent potential function for A^* as they may yield a potential that is not consistent: for an edge $e = (u,v)$, the distance from the cluster containing v to the cluster containing the target may be much smaller than the distance from the cluster of v to that of the target. In other words, the potential function $\pi : V \to \mathbb{R}$ induced by the cluster distances may satisfy $\pi(u) > w(e) + \pi(v)$, which results in a negative reduced weight $w_\pi(e) = w(e) - \pi(u) + \pi(v) < 0$ for e.

Method	Preprocessing			
	Time		Space	
Landmarks[1]	$\Theta(l \cdot D(n))$	++	$\Theta(l \cdot n)$	−−
Geometric Containers	$\Theta(n \cdot D(n))$	−−	$\Theta(m)$	∘
Edge Flags[2][3]	$\Theta(B \cdot D(n))$	−	$\Theta(k \cdot m)$ bits	−
PCD[2]	$\Theta(k \cdot D(n))$	+	$\Theta(k^2+B)$	++

Table 1: Speedups and preprocessing costs of goal-directed preprocessing heuristics. k denotes the number of clusters, l the number of landmarks, n the number of nodes, m the number of edges, B the number of border nodes, and $D(n)$ the running time of Dijkstra's algorithm. Plus signs ("+") mean a method is beneficial regarding the respective objective, a circle ("∘") or minus signs ("−") indicate average and bad performance, respectively.

2.3 Goal-Directed Preprocessing Heuristics

The following methods are goal-directed but perform some preprocessing unlike A^*. They are summarised in Figure 1; PCD is an algorithm of this category.

The concept of *landmarks* provides a lower bounding technique for A^* which is independent of node coordinates [8]. In a preprocessing step, a small number of landmark nodes is selected, and the distances $d(u, L)$ and $d(L, u)$ to and from each landmark L is computed for every node u. Then, for two nodes u and t, lower bounds for $d(u, t)$ are given by $d(L, t) - d(L, u)$ and $d(u, L) - d(t, L)$ for every landmark L by the triangle inequality. This is used in the query which follows the A^* algorithm. Using only 16 landmark nodes, a bidirectional implementation achieves a good speedup of 50 in terms of average number of settled nodes on a road network of about 6.7 million nodes. The preprocessing is fast since it performs only one shortest path search from each landmark (two for directed graphs). Still, sophisticated landmark selection strategies [9] can increase the preprocessing time significantly. Furthermore, though linear in the number of nodes for a constant number of landmarks, the additional space requirement is quite high since two distance values are stored for each node-landmark pair. The PCD algorithm may perform more preprocessing than landmarks, but it achieves goal-direction more space-efficiently through the adjustable number of clusters k.

If a graph is provided with a geometric layout, Dijkstra's algorithm can be directed using *geometric containers* requiring a linear amount of additional space: for each edge of the input graph, a geometric object is preprocessed that covers all nodes to which a shortest path starts with this edge. Edges not relevant for the target node are omitted in the query. High speedups of about 30 in terms of size of the search space are obtained even for simpler geometric objects such as bounding boxes [29]. Though showing good speedups, preprocessing geometric containers requires one single source shortest path search from every node, which is prohibitive for larger graphs.

An approach similar to geometric containers is based on the concept of *edge flags* [17, 16, 23]: the input graph is partitioned, and a flag is computed for each

[1] Preprocessing time does not include landmark selection and may increase depending on selection scheme [9].

[2] Preprocessing time does not include clustering.

[3] Space can be reduced through hierarchical clustering [23].

edge and each partition indicating whether the edge is contained in a shortest path to any node of this partition. The query only considers edges whose flag corresponding to the target region is set. The original approach used variants of grid partitioning [17], while applied to partitions obtained by Metis [21], the approach yields excellent speedups of up to 1 400 for a road network of about one million nodes and 225 partitions [16]. An extension to multiple levels further accelerates unidirectional search without increasing the space requirement [23]. Though lower than for geometric containers, the preprocessing time is still high since executing one shortest path search from every border node of every partition is necessary. For a road-network of 475 000 nodes and 100 partitions, this takes 2.5 hours [16].

2.4 Hierarchical Preprocessing Heuristics

Hierarchical speedup techniques exploit that shortest routes are largely restricted to smaller and smaller networks of important edges, the further away the search gets from source and target. During the original development of PCD, these methods were only beginning to show their full potential. Since then, hierarchical techniques have become very important since they, regarded as stand-alone methods, generally give higher speedups than goal directed techniques alone (e.g. [1, 6]). We refer to [3] for a detailed overview. However, note that using hierarchy has turned out to be mostly orthogonal to goal-directed techniques, so combinations suggest themselves as described in the following section. The fastest hierarchical technique now known is transit-node routing [1] and can be viewed as a descendant of PCD because it is also based on distance tables. The difference is that it computes actual distances between *important* nodes rather than lower bounds between clusters and uses the hierarchy of the network to reduce shortest path calculation to a small number of lookups in the distance table.

2.5 Combinations

Most preprocessing heuristics can be combined with bidirectional search, and also combining bidirected search with A^* can be beneficial [24, 13]. Both A^* and arc flags can be combined with hierarchical techniques to achieve improved speedup. We refer to [2] for a recent summary. Generally, we believe that replacing landmarks with PCD is promising for future work because it gives us a more flexible trade-off between preprocessing time, space, and query time.

2.6 Advantages of PCD over Related Methods

Similar to the edge flag approach and multi-level graph decomposition, the number of clusters k can be adjusted to decrease the costs of preprocessing. However, the preprocessing time for PCD is independent of the number of border nodes, and thus independent of the partitioning method, since exactly k single source shortest path computations are performed. In contrast, the edge flag approach requires one shortest path search from every border node of every cluster. Also, PCD achieves high speedups even for a simple clustering method such as grid clustering, while the performance of the separator-based multilevel method highly depends on the size of the separator. Furthermore, suitably choosing the

number of clusters allows an amount of preprocessed data sublinear in the input graph size unlike, for example, the landmark method. Moreover, PCD does not require graphs provided with a layout in contrast to geometric pruning, though partitions might still be obtained by techniques from computational geometry if a layout is given. The preprocessing costs and average speedups of PCD are compared to other goal-directed methods in Table 1.

3 Preliminaries

A *directed graph* G is a pair (V, E) of a finite set of *nodes* V and a set of *edges* $E \subseteq V \times V$, $n := |V|$ and $m := |E|$ denote their sizes, and G is called *sparse* if $m = \mathcal{O}(n)$. For an *undirected graph* the set of edges is given by $E \subseteq \{\{u, v\} \,|u, v \in V, u \neq v\}$. For a *weighted graph* each edge is assigned a *weight* by a function $w\colon E \to \mathbb{R}^{\geq 0}$. The *length* of a path P is denoted by $w(P)$, and the *distance* $d(s, t)$ between two nodes s and t is defined by the length of the shortest path from s to t.

Given a weighted graph $G = (V, E)$, $w\colon E \to \mathbb{R}^{\geq 0}$ and a *source node* $s \in V$, the *single-source shortest paths problem* is the problem of finding a shortest path from s to u for every $u \in V$. Additionally given a *target node* $t \in V$, the *single-source single-target shortest path problem* is the problem of finding the shortest path from s to t, which is clearly solved by solving the former problem.

Given a graph $G = (V, E)$, a collection $\mathcal{V} = \{V_1, \ldots, V_k\}$ of pairwise disjoint sets $V_i \subseteq V$, $1 \leq i \leq k$, such that $\bigcup_{i=1}^{k} V_i = V$ is called a *partition* of G and each set $V_i, 1 \leq i \leq k$ a *cluster* of G. The *size* $|V_i|$ of a cluster V_i is denoted by $s(V_i)$, and its *diameter* $\operatorname{diam}(V_i)$ defined by $\operatorname{diam}(V_i) := \max_{u,v \in V_i} d(u, v)$. Further, the *radius* $r(V_i)$ is defined by $r(V_i) := \min_{v \in V_i} \max(\max_{u \in V_i} d(v, u), \max_{u \in V_i} d(u, v))$. Note that $2\, r(V_i) \geq \operatorname{diam}(V_i)$.

A node $u \in V_i$ is called a *border node* of V_i if there is an edge $(u, v) \in E$ with $v \notin V_i$, and an *inner node* of V_i otherwise. The set of all border nodes of V_i is denoted by $B(V_i)$, and B denotes the total number of border nodes in a partitioned graph. A node $u \in V_j$, $j \neq i$, is called a *neighbour node* of V_i if there is an edge $(u, v) \in E$ with $v \in V_i$, and V_j is then called a *neighbour cluster* of V_i. The *distance* $d(V_i, V_j)$ between two clusters V_i and V_j is defined by $d(V_i, V_j) := \min_{u \in V_i, v \in V_j} d(u, v)$, and any path $p = (u, \ldots, v)$ with $u \in V_i$ and $v \in V_j$ is called a *shortest path from V_i to V_j* if $w(p) = d(V_i, V_j)$.

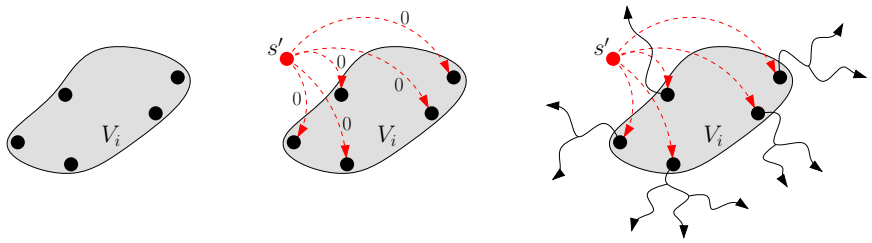
4 Preprocessing

Suppose, the input graph has been partitioned into clusters $V_1 \,\dot\cup \cdots \dot\cup\, V_k$. We want to compute a complete distance table that allows to look up

$$d(V_i, V_j) := \min_{s \in V_i, t \in V_j} d(s, t)$$

in constant time. We can compute $d(S, V_i)$ for a fixed cluster S and $i = 1, \ldots, k$ using just one single source shortest path computation: add a new node s' connected to all border nodes of S using zero weight edges. Perform a single source shortest path search starting from s'. Figure 4 illustrates this approach.

The following simple lemma shows that this suffices to find all connections from S to other clusters. The proof might be almost obvious, nevertheless, this



(a) Cluster V_i with its border nodes.

(b) Dummy node s' and zero-weight edges.

(c) Shortest paths search from s'.

Figure 4: Preprocessing connections from cluster V_i.

is an interesting result since several other speedup techniques require shortest path computations from *all* border nodes of *all* clusters.

Lemma 4.1 $d(S, V_i) = \min_{v \in V_i} d(s', v)$.

Proof We have $d(S, V_i) \leq \min_{v \in V_i} d(s', v)$ as any shortest path $(s', s, \ldots, v \in V_i)$ found during the search from s' contains a path $(s, \ldots, v)$ connecting the clusters S and V_i.

On the other hand, there cannot be a shorter connection from S to V_i. Assume the contrary, i.e., there is a path $(s \in S, \ldots, u \in S, u' \notin S, \ldots, v' \in V_i)$ with $d(s, v') < \min_{v \in V_i} d(s', v)$. Then $(s', u, \ldots, v')$ would constitute a shorter connection from s' to v', which is a contradiction. Repeating this procedure for every cluster yields the complete distance table. In addition, for each pair V_i, V_j we store a start point $s_{ij} \in V_i$ and an end point $t_{ij} \in V_j$ such that $d(s_{ij}, t_{ij}) = d(V_i, V_j)$.

5 Query

The PCD query algorithm generally follows Dijkstra's algorithm, additionally applying the preprocessed information. To allow sublinear execution time, the algorithm assumes that the distance values and predecessor information used by Dijkstra's algorithm have been initialized properly during preprocessing. In the following description of the bidirectional variant of the query, s and t denote the source and target node respectively, and S and T their clusters. A more detailed description including the unidirectional query can be found in [18]. The search works in two phases.

Algorithm Outline

In the first phase, we perform ordinary bidirectional search from s and t until the search frontiers meet, or until $d(s, s')$ and $d(t', t)$ are known, where s' is the first border node of S settled in the forward search, and t' the first border node of T settled in the backward search.

For the second phase we only describe forward search—backward search works completely analogously. The forward search grows a shortest path tree using Dijkstra's algorithm, additionally maintaining an upper bound $\hat{d}(s, t)$ for

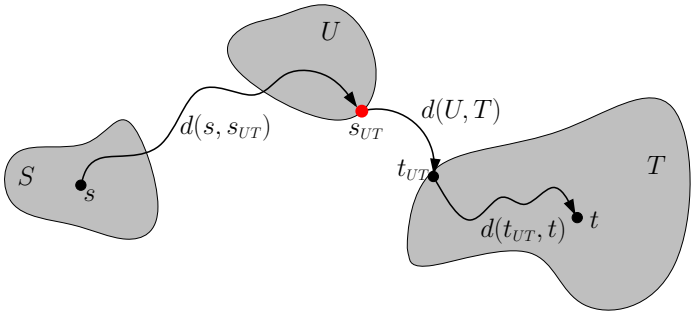


Figure 5: Updating the upper bound $\hat{d}(s,t)$ of $d(s,t)$ when settling the first node s_{UT} of a precomputed path from U to T.

$d(s,t)$, and computing lower bounds $\underline{d}(s,u,t)$ for the length of any path from s to t containing u. The search is pruned using the observation that any edge incident to u need not be considered if $\underline{d}(s,u,t) > \hat{d}(s,t)$. Phase two finishes when the search frontiers of forward and backward search meet. In a cleanup phase, the distance values and predecessor values changed during the search are reset to provide a proper initialization for the next query, which is efficiently done by maintaining a stack of all nodes visited during the search. It remains to explain how $\hat{d}(s,t)$ and $\underline{d}(s,u,t)$ are computed.

Upper Bounds

The upper bound is updated whenever a shortest path from s to a node $s_{UT} \in U$ is found such that u is the first node of a precomputed shortest connection between clusters U and T as illustrated in Figure 5. Then, a path from s to t—though not necessarily the shortest—has been discovered, and the upper bound is always updated to the length of the shortest such path found so far.

Lemma 5.1 *Let $s_{UT} \in U$, $t_{UT} \in T$ be the startnode-endnode pair of a precomputed shortest path from a cluster U to T and $d(U,T)$ denote its length. Then, $d(s,t)$ satisfies both the following conditions:*

$$d(s,t) \leq d(s,s_{UT}) + d(U,T) + d(t_{UT},t), \quad (1)$$
$$d(s,t) \leq d(s,s_{UT}) + d(U,T) + 2\,r(T). \quad (2)$$

Proof s and t are connected by a path $p = (s,\ldots,s_{UT},\ldots,t_{UT},\ldots,t)$ with length $w(p) = d(s,s_{UT}) + d(U,T) + d(t_{UT},t)$. Since $d(s,t) \leq w(p)$, condition (1) holds. Furthermore, $d(t_{UT},t) \leq \mathrm{diam}(T) \leq 2\,r(T)$, which proves inequality (2). The value of $d(s,u)$ has just been found by the forward search, and $d(U,T)$ and t_{UT} have been precomputed; thus, the sum in Equation (1) is defined if t_{UT} has already been found by the backward search. Otherwise, we use an upper bound of the diameter of T instead of $d(t_{UT},t)$ and Equation 2 applies.

Upper Bounds

In every settling step, a lower bound is estimated as shown in Figure 6.

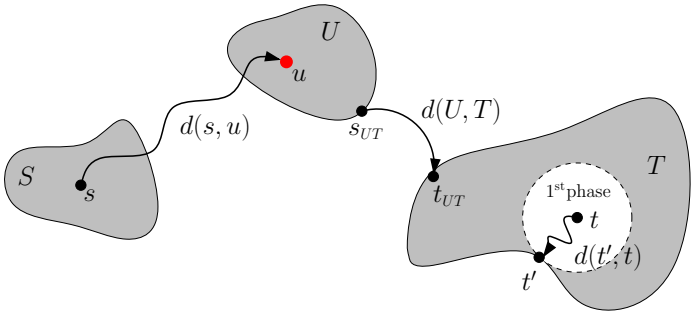


Figure 6: Estimating a lower bound $\underline{d}(s,u,t)$ for the length of any path from s to t containing u. The border node t' of T which is the closest to t has been determined by the end of the first phase.

Lemma 5.2 *Consider any node $u \notin T$, and let U denote its cluster. Then, the length of any path from s via u to t satisfies*

$$\underline{d}(s,u,t) := d(s,u) + d(U,T) + \min_{t' \in B(T)} d(t',t). \tag{3}$$

Proof We show that $\underline{d}(s,u,t) \leq d(s,u) + d(u,t)$, which is equivalent to showing $d(U,T) + \min_{t' \in B(T)} d(t',t) \leq d(u,t)$. Consider a shortest path $P = (u, \ldots, t'', \ldots t)$ from u to t where t'' denotes the first node on this path that is in cluster T. We have $d(u,t) = d(u,t'') + d(t'',t)$. Since $(u, \ldots, t'')$ is a connection from U to T we have $d(u,t'') \geq d(U,T)$. Furthermore, since t'' is a border node of T, we have $d(t'',t) \geq \min_{t' \in B(T)} d(t',t)$. The estimate in (3) can be computed efficiently as $d(s,u)$ has been found by forward search, U can be found by storing a cluster identifier with each node, $d(U,T)$ has been precomputed, and $\min_{t' \in T} d(t',t)$ has been determined in the first phase.

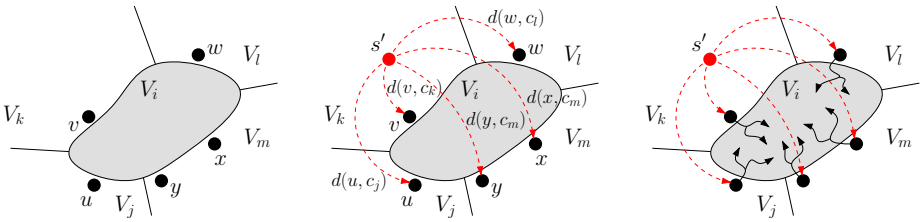
Robustness

It seems possible that the query algorithm keeps pruning nodes without inserting new nodes so that the priority queues run empty in the end. However, this will not happen since nodes on the shortest path are never pruned as shown in Lemma 5.3.

Lemma 5.3 *The PCD query algorithm never prunes a node on the shortest path.*

Proof Let SP be the shortest path from s to t and let $u \in V$ be any node of SP. If $d(u,t) \leq \min_{v \in B(T)} d(v,t)$, u will be settled in the backward search already in the first phase; further, it will not be considered for pruning in the opposite direction in the second phase anymore.

Now assume that $d(s,u) > \min_{v \in B(S)} d(s,v)$ and $d(u,t) > \min_{v \in B(T)} d(v,t)$. As $w(SP) = d(s,t)$, $d(s,t) \geq \underline{d}(s,u,t)$ is satisfied by Lemma 5.2. On the other hand, $d(s,t) \leq \hat{d}(s,t)$ at any time by Lemma 5.1, so $\underline{d}(s,u,t) \leq \hat{d}(s,t)$. The immediate consequence from this is that the PCD query algorithm never runs dry, and the search fronts eventually meet. The path found when finishing not only is a valid path from the source to the target, it also is the shortest, which follows from the correctness of Dijkstra's algorithm.



(a) Cluster V_i with neighbour clusters and their border nodes u to y.

(b) Dummy s' and edge weights corresponding to the centers' distances.

(c) Redistributing V_i by a single source shortest paths search from s'.

Figure 7: Deleting cluster V_i by distributing its member nodes to the neighboured clusters.

Space Efficient Implementation

The query algorithm described above is straight forward to implement using space $\mathcal{O}(k^2 + n)$, which can be improved to $\mathcal{O}(k^2 + B)$. The problem is that when settling a node u, we need to know its cluster id. The key observation is that clusters only change at border nodes so that it suffices to store the cluster ids of the B border nodes in a hash table.

6 Partitioning

Three different graph partitioning methods are introduced in the following. For the sake of simplicity, graphs are assumed undirected in this section, but the methods can be extended to directed graphs. In particular, the preprocessing and query algorithms of PCD do *not* require undirected graphs.

6.1 k-Center Clustering

For any set $C = \{c_1, \ldots, c_k\} \subseteq V$ of k distinct *centers*, assigning each node $v \in V$ to the center closest to it results in a *k-center clustering*. In connection with k-center clustering, the *radius* $r(V_i)$ of a cluster V_i denotes the distance from its center c_i to the furthest member. Note that $2\,r(V_i)$ is an upper bound of $\mathrm{diam}(V_i)$ for this definition of radius too. A k-center clustering can be obtained using *k'-oversampling*: a sample set C' of k' centers is chosen randomly from V for some $k' \geq k$, and a k'-center clustering is computed for it by running one single source shortest path search from a dummy node connected with each center by a zero-weight edge. Then, clusters are deleted successively until k clusters are left.

A cluster V_i is deleted by removing the corresponding center c_i from C' and reassigning each member of V_i to the center now closest to it (see Fig. 7). This amounts to a shortest path search from the neighboring clusters which now grow into the deleted cluster. This process terminates with a $(k'-1)$-clustering. There are several ways to choose a cluster for deletion: in Section 7 results are shown for the MinSize and the MinRad heuristics, which choose the cluster of minimum size and minimum radius respectively, and the MinSizeRad heuristic,

which alternates between the former two.

Calculating the initial k'-center clustering affords a time of $D(n)$, and deleting a cluster V_i takes $D(s(V_i))$ where $D(()x)$ denotes the time needed for settling x nodes using Dijkstra's algorithm for a sparse graph. The cluster size is bounded by $s(V_i) \leq n-k'+i = \mathcal{O}(n)$ in the i-th deletion step, $i \in \{1, \ldots, k'-k\}$, so the time of calculating a k-center clustering by k'-oversampling amounts to $\mathcal{O}((k'-k)\, D(n))$, which is $\mathcal{O}(k \lg k\, D(n))$ for $k' := k \lg k$.[4] If the MinSize heuristic is used, the cluster size can be bounded by $s(V_i) \leq \frac{n}{k'-(i-1)}$ in the i-th step, $i \in \{1, \ldots, k'-k\}$. This allows improving the running time to $\mathcal{O}(n \lg n \lg \frac{k'}{k})$, which becomes $\mathcal{O}(n \lg n \lg\lg k)$ for $k' := k \lg k$ and $D(n) = \mathcal{O}(n \lg n)$:

$$\mathcal{O}\left(\sum_{i=k+1}^{k'} D\left(\frac{n}{i}\right)\right) = \mathcal{O}\left(\sum_{i=k+1}^{k'} \frac{n}{i} \lg \frac{n}{i}\right) = \mathcal{O}\left(\sum_{i=k+1}^{k'} \frac{n}{i} \lg n - \underbrace{\sum_{i=k+1}^{k'} \frac{n}{i} \lg i}_{\geq 0}\right)$$

$$= \mathcal{O}\left(n \lg n \left(\sum_{i=1}^{k'} \frac{1}{i} - \sum_{i=1}^{k} \frac{1}{i}\right)\right) = \mathcal{O}\left(n \lg n \left(\lg k' - \lg k\right)\right) = \mathcal{O}\left(n \lg n \lg \frac{k'}{k}\right)$$

Thus, the partitioning algorithm has negligible cost compared to computing cluster distances, which requires searching $\mathcal{O}(nk)$ nodes. [18] contains a more detailed description of k-center clustering.

The radius of a cluster affects the lower bounds of its members, and it seems that a good partition for PCD has clusters of similar size and a low average radius. Oversampling indeed keeps a low average radius as deleted clusters tend to be distributed to neighbors of lower radius. But, a higher radius is acceptable for smaller clusters since the lower bound is not worsened for too many nodes then, whereas a low radius allows a bigger size. Both values can be combined into the *weighted average radius*, where single radii are weighted by their clusters' sizes.

Our k-center heuristics are compared with a simple partitioning based on a rectangular grid and with Metis [21]. Metis was originally intended for parallel processing where clusters should have close to equal size and small boundaries in order to reduce communication volume.

6.2 Grid Clustering

A quite simple approach of graph partitioning is to arrange a clustering in a grid-like manner: a square grid is applied above the graph, and all nodes in the same square together form a cluster. This method needs node coordinates, and the graph instances used for testing indeed contain this information. However, this is the only point where they are used.

To achieve a grid clustering with (at least) k clusters, the side length l of a square has to be chosen carefully. W.l.o.g. let $x_{min} = 0$, $x_{max} = 1$, $y_{min} = 0$, and $y_{max} = c$ be the minimum and maximum x- and y-coordinates respectively of all nodes in a given graph. Now, put the side length $\ell = \sqrt{\frac{c}{k}}$, the number of squares in the horizontal direction $a = \left\lceil \frac{1}{\ell} \right\rceil$, and the number of squares in the vertical direction $b = \left\lceil \frac{a}{\ell} \right\rceil$. With this choice all nodes will be covered since

[4] Throughout this paper $\lg x$ stands for $\log_2 x$.

$a\,\ell \geq 1$ and $b\,\ell \geq c$, and sufficiently many clusters will be achieved since $a\,b \geq k$. Note that empty squares might occur, which would yield empty clusters and thus can be deleted. For a graph with n nodes featuring geographic coordinates, a grid clustering can be calculated in time $\Theta(n)$.

6.3 Metis

An approach much more sophisticated than grid clustering is provided by the software package Metis [21], which is originally intended for balancing computations in parallel processing while minimising the communication volume. It contains a variety of programs and libraries for partitioning graphs, particularly for k-way partitioning. A *k-way partition* of a graph $G = (V, E)$ is a partition $\mathcal{V}$ of k clusters with the additional property that $|U| = \frac{n}{k}$ for every cluster $U \in \mathcal{V}$. The relevant Metis procedure follows a multilevel k-way partitioning scheme described in [14], and finds a clustering of k equal size regions (not necessarily contiguous), minimising the edge cut, where the *edge cut* of a partition denotes the number of edges with end nodes in different clusters.

The clusters obtained by Metis are of perfectly uniform size. On the other hand, this is paid for with highly differing diameters, which is further added to by possible non-contiguous clusters. The resulting effects on the query are examined in Section 7.4.

7 Experiments

The performance of the PCD algorithm is examined experimentally and compared to that of Dijkstra's algorithm in terms of query time and size of the search space. Here, *search space* refers to the subset of nodes settled in a single query. The main value to specify this comparison is *speedup*, which denotes the ratio of a measured value of Dijkstra's algorithm to that of PCD and refers to either query time or number of settled nodes. All numbers presented in this section are average values over 1 000 shortest path queries, each for a random pair of source and target node.

7.1 Instances

All test instances used for the experiments presented in this paper represent real-world road networks, whose names and sizes can be found in Table 2 for the European and Table 3 for the US road network. In these graphs, edges correspond to road segments, and nodes represent junctions of segments. For every instance, two different edge weight functions are available, one corresponding to average travel times, the other to travel distances.

The graphs listed in Table 2 represent the road network of 14 Western European states. The original data for this—which was made available for scientific use by PTV AG—contains one length value for each edge, which provides the distance function mentioned above. Furthermore, each edge is assigned one out of 13 road categories, ranging from forest roads to fast motorways. To each category, an average travel speed between 10 km/h and 130 km/h is assigned, which yields the average travel time of each edge. Furthermore, some edges represent ferry connections for which the average travel times are contained in the

Instance	n	m	Description
CEN	5,342,276	6,664,561	Belgium, Danmark, Germany, Luxembourg, the Netherlands
WSO	4,190,068	5,251,527	Austria, France, Switzerland, Italy
SCA	2,453,610	2,731,129	Norway, Sweden
GBR	2,149,793	2,590,401	Great Britain
IBE	872,083	1,177,734	Spain, Portugal
DEU	4,375,849	5,483,579	Germany
SUI	630,962	771,694	Switzerland

Table 2: The graph instances of the European road network used for the experiments.

Instance	n	m	Description
SOU	7,588,996	8,851,170	South (AL, AR, FL, GA, KY, LA, MS, MO, NC, SC, TN, VA, WV)
MID	5,246,822	6,494,670	Midwest (IL, IN, IA, KS, MI, MN, NE, ND, OH, SD, WI)
WES	4,429,488	5,296,150	West (CA, CO, ID, MT, NV, OR, UT, WA, WY)
SWT	3,561,925	4,382,697	Southwest (AZ, NM, OK, TX)
MAT	2,226,138	2,771,948	Middle Atlantic (DC, DE, MD, NJ, NY, PA)
NEN	896,115	1,058,481	New England (CT, ME, MA, NH, RI, VT)

Table 3: The graph instances of the US road network used for the experiments.

input data. The graphs DEU and SUI are subgraphs of CEN and WSO respectively.

The graphs listed in Table 3 represent the road network of the 48 contiguous states of the United States and the District of Columbia, which was obtained from [28]. The original data also contains a road category for each edge as well as a length value, from which an average travel time is determined in the same way as above. However, there are only four different categories ranging from local roads to primary highways, to which average speeds between 40 km/h and 100 km/h are assigned.

All instances are undirected, connected, and do not have any self-loops or parallel edges. As mentioned before, undirected graphs are only used for the sake of simpler partitioning—the precomputing and query algorithms work for directed graphs too. The information about the direction of each edge in the original data for the European instances is ignored, whereas no direction is available for the US instances. Furthermore, the original data provides geographic coordinates for each node; this information is only used for estimating the grid clustering described in Section 6.2. The instances used are generated in the same way as in [25], in particular, the same average speeds are assigned to the road categories when deriving the travel times.

The graphs reflect the following characteristic of the real-world road networks they represent: while a higher amount of junctions in urban areas yields shorter road segments, the distances between intersections are typically higher in rural areas. Therefore, the terms *urban* and *rural* will be used to refer to partitions of the graphs with lower and bigger edge weights respectively. This notion is independent of the graphs' geometries, particularly of the node coordinates.

7.2 Experimental Setup

All experiments presented in this section were performed on an AMD Opteron clocked at 2.4 GHz with 8 GB of main memory running Linux. The PCD algorithm has been implemented in C++ using the data structure 'static_graph' of the C++ library LEDA 5.1 [20] and compiled with the GNU C++ compiler g++-3.4 with optimisation level `-O3`.

7.3 Query Performance

k-center clustering with $(k \lg k)$-oversampling using the MinSize deletion heuristic serves as the basis clustering method and is used for all experiments of this section. This method outperforms the other clustering methods, which is shown in Section 7.4. Moreover, all results refer to the bidirectional version of PCD unless otherwise stated.

Table 4 summarises the values measured for the PCD query algorithm comparing different instances for the same number of clusters $k = 2^{10}$, while results for varying k and the fixed instance DEU are shown in Table 5. The highest speedup measured is 114.9 in terms of average query time, achieved for the graph DEU with travel time edge weights. Moreover, speedups of more than 30 are still achieved for a small preprocessing time and a number of border nodes and cluster pairs $(B + k^2)$ significantly smaller than n.

Graph	k	$\frac{B+k^2}{n}$	prep. [min]	Bidirectional PCD Query			
				t [ms]	*spd*	settled [♯]	*spd*
CEN	2^{10}	0.22	144.3	188	34.5	161,597	20.1
WSO	2^{10}	0.27	92.3	175	21.3	176,514	14.2
SCA	2^{10}	0.44	60.7	81	36.5	70,766	22.9
GBR	2^{10}	0.51	43.6	53	39.6	54,727	25.7
IBE	2^{10}	1.25	11.7	20	24.7	26,591	20.4
DEU	2^{10}	0.26	123.0	157	35.0	127,604	20.2
SUI	2^{10}	1.71	11.1	9	31.1	12,848	31.4
SOU	2^{10}	0.15	160.2	319	20.5	299,202	13.9
MID	2^{10}	0.22	89.2	223	18.5	242,153	12.8
WES	2^{10}	0.25	80.8	169	19.0	159,409	14.4
SWT	2^{10}	0.31	60.9	137	21.3	126,383	15.5
MAT	2^{10}	0.50	35.5	76	21.1	83,577	15.2
NEN	2^{10}	1.21	11.0	28	19.3	34,625	15.0

Table 4: Performance of PCD for several graphs with travel time edge weights using the same number of clusters $k = 2^{10}$. The following abbreviations are used: 'prep.' denotes the preprocessing time (including partitioning), 't' the average query time, 'settled' means the average number of settled nodes, and '*spd*' the corresponding average speedup.

Graph	k	$\frac{B+k^2}{n}$	prep. [min]	Bidirectional PCD Query			
				t [ms]	*spd*	settled [♯]	*spd*
DEU	2^4	< 0.01	2.6	2114	2.5	1,028,720	2.4
	2^5	< 0.01	6.5	1631	3.3	833,545	3.1
	2^6	0.01	11.1	971	5.2	553,863	4.3
	2^7	0.01	19.1	622	7.7	404,121	5.8
	2^8	0.03	35.0	422	12.3	295,525	8.5
	2^9	0.08	68.6	242	20.9	188,239	13.2
	2^{10}	0.26	123.0	157	35.0	127,604	20.2
	2^{11}	0.99	246.9	105	60.3	82,404	32.7
	2^{12}	3.88	558.2	62	**114.9**	50,417	57.4

Table 5: Performance of PCD for varying numbers of clusters k measured on DEU with travel times. The same abbreviations as in Table 4 are used.

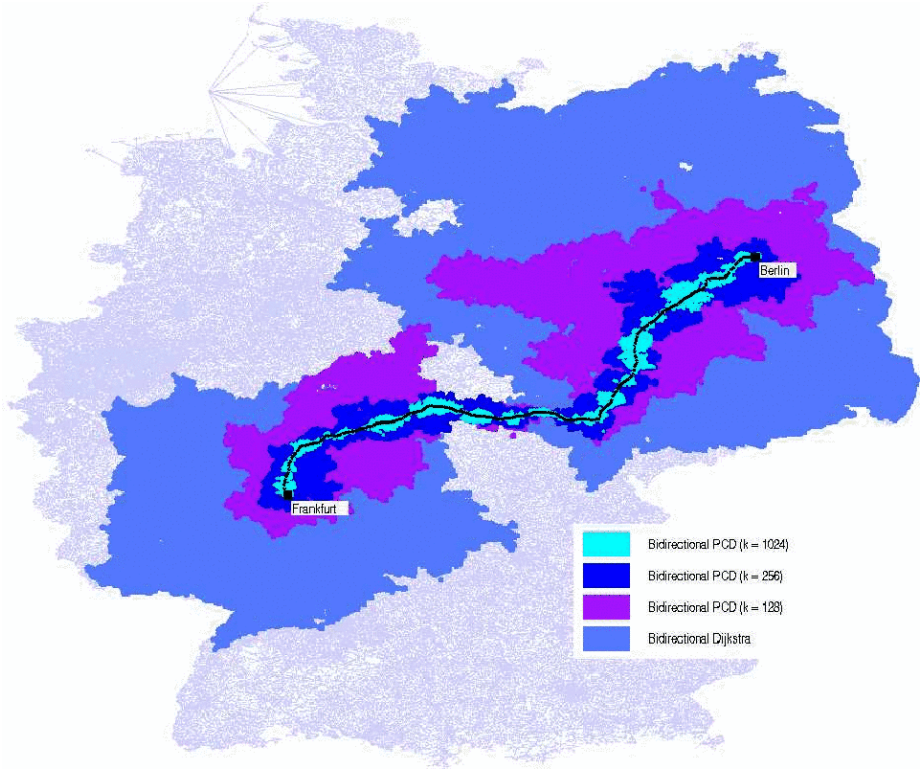


Figure 8: Search spaces of a sample query from Frankfurt to Berlin for the graph `DEU` with travel time edge weights and different values of k.

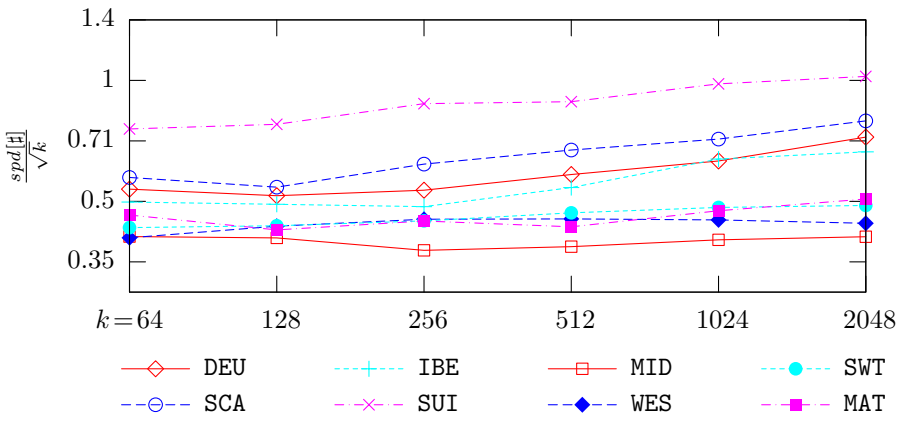


Figure 9: Performance of PCD for several instances and varying values of k with travel time edge weights. The speedups in terms of number of settled nodes are scaled by $\sqrt{k}$.

7.3.1 Search Space and Query Time

Figure 8 compares the shape of the area explored by a bidirectional Dijkstra search to that of PCD for selected values of k. Dijkstra's algorithm roughly searches two touching balls centered at the source and target node respectively.

In contrast, the search space of PCD describes a narrow corridor surrounding the shortest path from the source to the target. This corridor approximately covers those clusters which contain one or more nodes of the shortest path since nodes outside these clusters are mostly pruned: when crossing the border to a cluster outside the corridor, there is a jump in the lower bound estimate as a different precomputed shortest path is used for the estimation.

The number of these clusters covering the shortest path roughly doubles if k is multiplied by four, while the average cluster size is $\frac{n}{k}$, so PCD roughly explores $\mathcal{O}(\sqrt{k})$ clusters visiting $\mathcal{O}(\frac{n}{\sqrt{k}})$ nodes on the average. Dijkstra's algorithm visits an average number of nodes of $\mathcal{O}(n)$ independent of k, which suggests the speedup of PCD towards Dijkstra is $\mathcal{O}(\sqrt{k})$. This heuristic consideration is supported by Figure 9: plotting the average speedup values in terms of number of settled nodes, scaled by $\sqrt{k}$, against the number of clusters k yields nearly-flat lines.

Figure 8 also illustrates the gap in the lower bound estimate mentioned in Section 1: in this example, reducing the number of clusters to $k = 128$ causes a second corridor to be explored around a path slightly longer than the shortest. Because the lower number of clusters results in less exact lower bound estimates, nodes close to this nearly-shortest path are not pruned. Apparently, the first gap in the lower bound estimate mentioned in Section 1 is greater than the difference in length between these two paths for $k = 128$ but not for $k = 256$.

Going back to Table 4 and 5, the measured speedups in terms of average query time are generally higher than those in terms of average number of settled nodes. A minor reason for this is that nodes pruned after settling contribute to the size of the search space, but take only little time since their adjacent edges are not explored.

However, the main reason lies in the sizes of the priority queues, on which the execution times for queue operations depend. In Dijkstra's algorithm the priority queues keep growing until the search frontiers meet. In contrast, PCD searches two small balls around the source and the target, pruning the nodes on their boundaries shortly after. The search frontiers turn into the small corridor mentioned above, and the corresponding priority queues both hold a number of nodes remaining roughly constant until finishing. Since the average queue size of PCD is smaller than that of Dijkstra's algorithm, the average speedup in terms of query time is higher than in terms of number of settled nodes. This relation is analyzed in more detail in [18].

7.3.2 Travel Times vs. Travel Distances

The speedups achieved for travel times are generally higher than those for travel distances (see Table 6). This is because edges away from the *quickest* path (i.e. the shortest path w.r.t. the travel time edge weights) have relatively high travel time values compared to edges on the quickest path since most of the latter represent major roads allowing a high travel speed. If the quickest path is left in a query, pruning therefore happens after fewer steps for travel times compared

Graph	k	Bidirectional PCD Query			
		t [ms]	*spd*	settled [♯]	*spd*
CEN	2^{10}	404	9.5	368,704	7.7
WSO	2^{10}	269	8.5	313,208	7.3
SCA	2^{10}	108	12.0	131,543	10.3
GBR	2^{10}	182	9.0	156,321	8.0
IBE	2^{10}	35	10.8	47,903	10.3
DEU	2^{10}	275	10.0	294,779	7.9
SUI	2^{10}	36	10.9	40,351	9.8
SOU	2^{10}	475	10.0	501,127	8.0
MID	2^{10}	424	9.0	384,047	7.4
WES	2^{10}	200	11.1	249,061	9.4
SWT	2^{10}	187	10.5	226,929	8.6
MAT	2^{10}	106	10.2	137,903	8.4
NEN	2^{10}	48	9.6	64,835	8.3

Table 6: Performance of PCD for several graph instances with travel distance edge weights using the same number of clusters $k = 2^{10}$. The abbreviations are explained in Table 4.

to leaving the shortest path using distances. PCD visits fewer nodes for travel time edges weights, while the size of the search space for Dijkstra's algorithm is similar for both weight functions.

This dependency on the edge weight function is even more significant for the speedups in terms of average query times. The reason for this lies in the smaller average size of the priority queue in Dijkstra's algorithm. This is bigger if travel times are used since they give the shortest path search some flavour of depth first search, while travel distances show more similarity to breadth first search. Furthermore, the speedups measured with travel time edge weights are generally lower for the US instances compared to the European. The less hierarchical road classification in the input data of the US (see Section 7.1) has an effect similar to using travel distances. Nevertheless, the values are still higher than those for travel distance edge weights.

7.3.3 Bidirectional vs. Unidirectional

The unidirectional query algorithm traverses the corridor only in the direction from s to t, while it searches a small ball around the target additionally without any pruning. The smaller the distance between t and the border of its cluster T, the more nodes outside this cluster are explored in the backward search until T is fully explored. In contrast, the bidirectional algorithm may prune nodes in T, particularly if a beneficial position of s in S causes a small gap in the lower bound estimates of the backward direction.

Figure 10 illustrates the relation between the uni- and bidirectional query algorithm in terms of average number of settled nodes. For travel distances, this relation is slightly greater than one, which could be expected from the consideration above, while it goes up to two for travel times. This can be explained in the following way: for travel times the search may follow major roads—which usually have low edge weights—and run far outside T after few

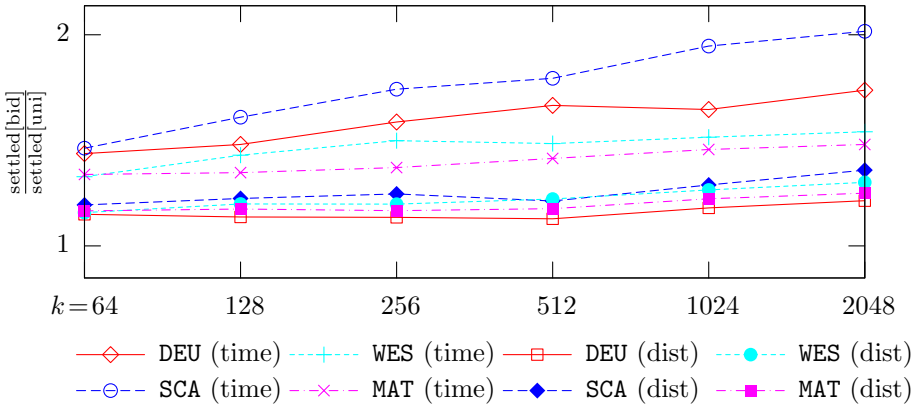


Figure 10: Relation between number of settled nodes in the unidirectional and bidirectional PCD query algorithm for several instances and both edge weight functions.

steps. Hence, the backward search might already settle many nodes outside T before all nodes of T on side roads are settled and the backward search stops. Figure 10 also shows that this is less significant for the US instances: due to the less hierarchical road classification the plots for `WES` and `MAT` are below the two European examples (but still above the plots for travel distances).

Furthermore, the relation rises with increasing k in all cases: the more clusters there are, the earlier and more often the upper bound is updated, which takes stronger effect in the bidirectional query.

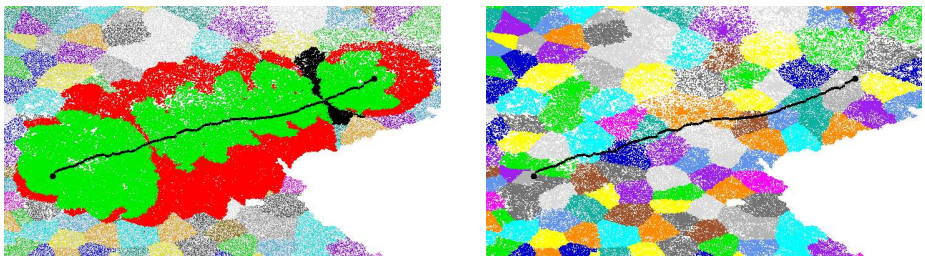
To sum up, the unidirectional algorithm is affected stronger by an adverse position of t in T, but cannot profit from the position of s in S unlike the bidirectional version. The example shown in Figure 11 further illustrates this issue: the source node's position close to its cluster's border causes the backward search to explore only a narrow corridor, while the unidirectional algorithm traverses a wider corridor from left to right due to the adverse position of the target. Moreover, there is a large ball around the target for the unidirectional algorithm due to the target's position close to the border.

7.4 Partitioning Methods

All results presented so far use k-center clustering with $(k \lg k)$-oversampling using the MinSize deletion heuristic for partitioning the input graph. In the following this is compared with results for variants of k'-oversampling and the other partitioning methods from Section 6. They all turn out to perform worse. All speedup values in this section are measured on the graph `DEU` using travel time edge weights, while oversampling is performed with $k' = k \lg k$ unless otherwise mentioned.

7.4.1 Oversampling vs. Grid and Metis

Figure 12 compares the average speedups of several heuristics for $(k \lg k)$-oversampling as well as grid clustering and Metis. Both Metis and the



(a) The green area in the interior is explored by both, the surrounding red area only by the unidirectional, and the black strip at the right only by the bidirectional version.

(b) Position of the source node (left) and the target node (right) in their respective clusters.

Figure 11: Search spaces of the unidirectional and the bidirectional algorithm for a sample source-target pair.

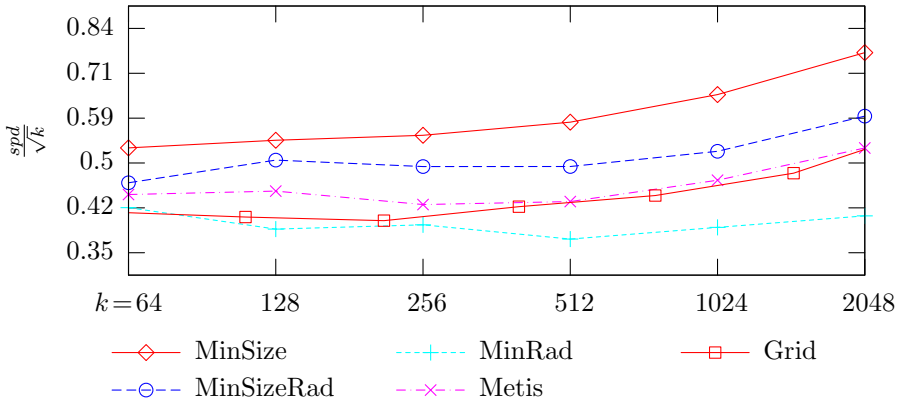


Figure 12: Performance of PCD depending on the partitioning method.

simple grid approach perform better than the MinRad heuristic, but yield average speedups still smaller than those for MinSizeRad. All methods are well outperformed by the MinSize deletion heuristic.

At first sight low cluster diameters improve the speedups since they improve the lower bound estimates, and oversampling with the MinSize deletion heuristic indeed shows the lowest average cluster radius. On the other hand, the average radius does not fully reflect the quality of a partition. Oversampling may generally yield fairly low diameters since the nodes of a deleted cluster tend to be distributed to neighbours with smaller radii. However, the fairly low average radius for the MinRad heuristic and also for Metis suggests much better speedups, particularly compared to grid clustering.

The following insight helps explaining this: it seems that an ideal partition should have clusters of the same size and a very small diameter. Nevertheless, a higher diameter is acceptable for clusters with a small number of nodes. In such a case only few nodes are affected by the poor lower bound resulting from their cluster's big diameter. Analogously, a low diameter allows a bigger cluster size.

That is what the MinSize heuristic exactly implements: just after oversampling, i.e. before deletion has started, all clusters have a very small size, while their radii differ a lot, with lower values in urban and bigger radii in rural areas. MinSize now picks clusters for deletion regardless of their radius, so deletion happens in all areas of the graph unlike for MinRad (see below). As mentioned before, the way of redistributing deleted clusters causes high-diameter clusters to grow less in size than clusters with low diameter. MinSize produces partitions with a non-zero standard deviation of the size and a relatively high deviation of the radii. (See [18] for deviation of cluster radii and sizes from average of measured values.) The crucial point is that clusters with a great radius are small in size, and the big-sized clusters have a low radius.

In contrast, the MinRad heuristic mostly selects clusters in urban areas for deletion and therefore grows urban clusters in size and diameter. This results in similar diameters but highly differing sizes, so the urban clusters—in which most most of the nodes are contained—have a bad relation between size and diameter. The opposite holds for Metis: it yields a zero-standard deviation of the sizes, which must be paid for with some clusters of very high radius, which explains the moderate query performance of this method.

7.4.2 Choice of k' for k'-Oversampling

The performance of k'-oversampling depends on the value of k', which is illustrated in Figure 13. In this figure the value $\Delta = k \lg k - k$ denotes the gap between $k \lg k$ and k. Starting from $k' = k$, which means just choosing k centers randomly, even a small value of $\Delta > 0$ causes considerably higher speedups. Further increasing k' also increases the measured speedups, until values higher than $k \lg k$ show no significant improvement.

7.5 Border Nodes and Additional Space

Apart from the precomputed cluster distances, the cluster ids add to the space for precomputed data and must be stored for the B border nodes. As illustrated in Figure 14, this number B depends on the partitioning method, of which Metis

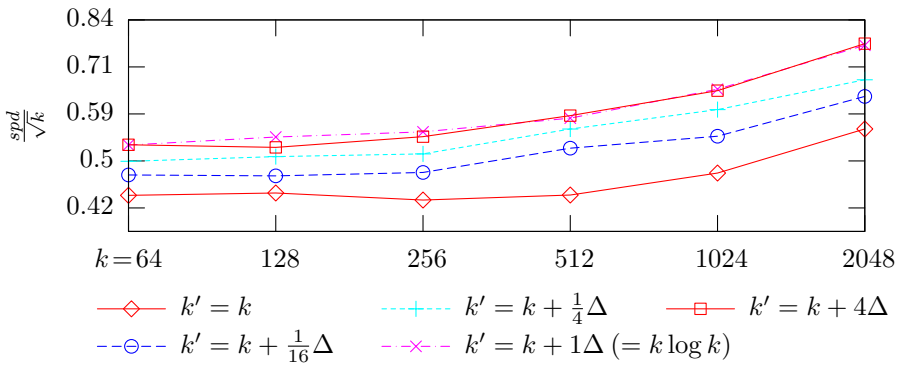


Figure 13: Results for k'-oversampling with MinSize deletion depending on the value of k'.

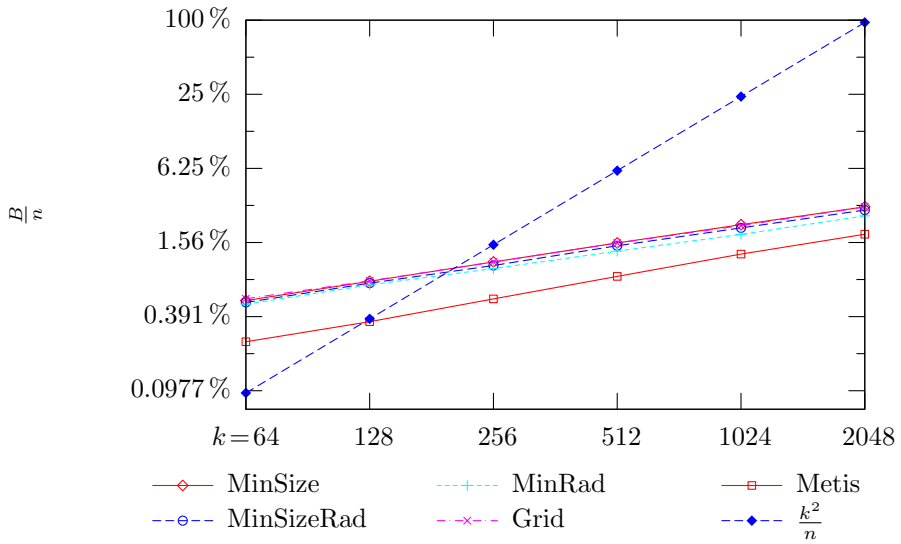


Figure 14: Relative number of border nodes depending on the method of partitioning. The numbers refer to `DEU` with travel time edge weights.

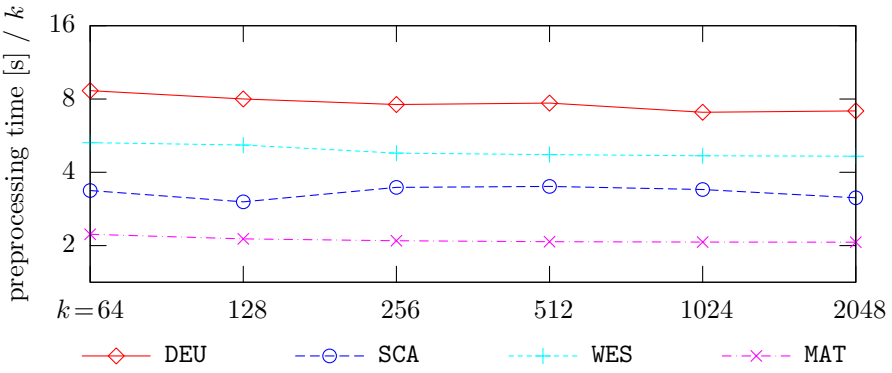


Figure 15: Time required for precomputing the cluster distances (scaled by the number of clusters k) using travel time edge weights.

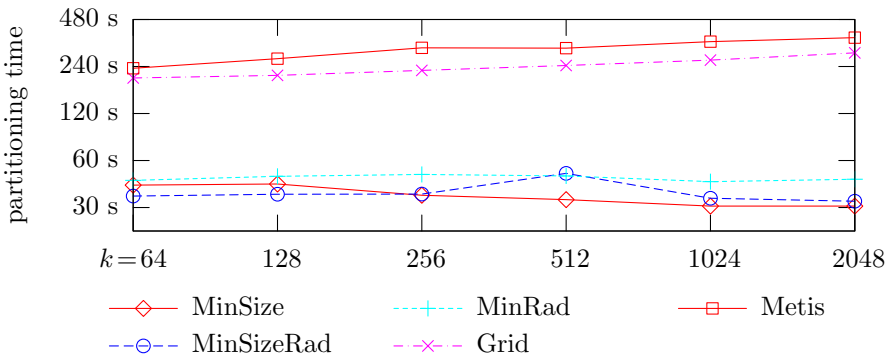


Figure 16: Partitioning time for several methods and varying k, measured for DEU using travel time edge weights.

yields the fewest border nodes since one of its objectives is reducing the edge cut (see Section 6.3). However, independent from the method, the amount of border nodes is negligible compared to k^2 for larger values of k since $B = \mathcal{O}(\sqrt{k})$. For lower k, the value of $B + k^2$ is very small anyway (also see Table 5), so the number of border nodes B and therefore the partitioning method does not affect the space requirement significantly. This holds for oversampling in particular, further supporting its use as the reference method for partitioning in Section 7.3.

7.6 Preprocessing Time

Figure 15 shows the time needed for precomputing the cluster distances scales with k as expected from Section 4. Though DEU and WES are of similar size (see Tables 2 and 3), their preprocessing times differ by a factor of about two, which holds for SCA and MAT analogously. Similar to the different speedups in terms of query time as explained in Section 7.3, the less hierarchical road-classification of the US instances causes a smaller size of the priority queue and thus a faster preprocessing.

The partitioning times for several methods are presented in Figure 16: the

oversampling heuristics need less than one minute for the fairly big graph instance `DEU`, again supporting the role as the basis clustering technique throughout this section. Though MinSizeRad shows a somewhat unspecific behaviour, MinSize works slightly faster than MinRad as expected from Section 6. Compared to oversampling, which estimates cluster radii while partitioning, the times for grid clustering and Metis are relatively high since they compute the radii in an additional step. Still, the partitioning time is negligible compared to the time needed for computing the cluster distances for all tested partitioning methods.

8 Conclusion

We have demonstrated that PCD can give route planning in road networks a strong sense of goal direction leading to significant speedups compared to Dijkstra's algorithm using only sublinear space. The most obvious task for future work is to combine PCD with hierarchical speedup techniques, e.g. with contraction hierarchies [6], which are both simple and provide very good speedups with little preprocessing time. Experiments in [2] indicate that goal directed techniques such as landmark A^* can be applied to a contracted version of the input graph for which further contraction would not be useful. This kind of combination thus yields a more robust technique for graphs which exhibit less hierarchy than road networks. PCD might be an alternative to landmark A^* which needs less space. On the other hand, hierarchical techniques can be used to quickly compute cluster distances, i.e., for large graphs we are no longer constrained by the invested preprocessing time but only the space consumption of the table of cluster distances [15, 6].

PCD itself could be improved by giving better upper and lower bounds. Upper bounds are already very good and can be made even better by splitting edges crossing a cluster border such that the new node has equal distance from both cluster centers. For example, this avoids cluster connections that use a small road just because the next entrance from a motorway is far away from the cluster border. While this is good if we want to approximate distances, initial experiments indicate that it does not give additional speedup for exact queries. The reason is that *lower bounds* have a quite big error related to the cluster diameters. Hence, better lower bounds could lead to significant improvements. For example, can one effectively use all the information available from precomputed distances between clusters explored during bidirectional search?

It seems that a good partitioning algorithm should look for clusters of about equal size and low diameter. These might be two of the main parameters for an easily computable objective function whose value indicates whether a clustering might be suitable for PCD. In the literature there is a lot of work on approximation algorithms for various k-center problems. It would be interesting to adapt some of the proposed algorithms to our situation, both in order to develop an objective function and to further improve the query speedup.

References

[1] H. Bast, S. Funke, P. Sanders, and D. Schultes. Fast routing in road networks with transit nodes. *Science*, 316(5824):566, 2007.

[2] R. Bauer, D. Delling, P. Sanders, D. Schieferdecker, D. Schultes, and D. Wagner. Combining hierarchical and goal-directed speed-up techniques for Dijkstra's algorithm. In *7th Workshop on Experimental Algorithms (WEA)*, volume 5038 of *LNCS*, pages 319–333. Springer, 2008.

[3] D. Delling, P. Sanders, D. Schultes, and D. Wagner. Engineering route planning algorithms. submitted for publication, `http://i11www.ira.uka.de/extra/publications/dssw-erpa-09.pdf`, 2008.

[4] E. W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1:269–271, 1959.

[5] Michael L. Fredman and Robert E. Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *Journal of the ACM*, 34(3):596–615, 1987.

[6] R. Geisberger, P. Sanders, D. Schultes, and D. Delling. Contraction hierarchies: Faster and simpler hierarchical routing in road networks. In *7th Workshop on Experimental Algorithms (WEA)*, volume 5038 of *LNCS*, pages 319–333. Springer, 2008.

[7] Andrew V. Goldberg. A simple shortest path algorithm with linear average time. In *Proceedings of the 9th Annual European Symposium on Algorithms (ESA-01)*, volume 2832 of *LNCS*, pages 230–241. Springer, 2001.

[8] Andrew V. Goldberg and Chris Harrelson. Computing the shortest path: A^* search meets graph theory. In *Proceedings of the 16th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA-05)*, pages 156–165. SIAM, 2005.

[9] Andrew V. Goldberg and Renato F. Werneck. Computing point-to-point shortest paths from external memory. In *Proceedings of the 7th Workshop on Algorithm Engineering and Experiments (ALENEX-05)*, pages 26–40. SIAM, 2005.

[10] Torben Hagerup. Sorting and searching on the word RAM. In *Proceedings of the 15th Annual Symposium on Theoretical Aspects of Computer Science (STACS-98)*, pages 366–398, London, UK, 1998. Springer.

[11] P. E. Hart, N. J. Nilsson, and B. Raphael. A formal basis for the heuristic determination of minimum-cost paths. *IEEE Transactions of Systems Science and Cybernetics*, SSC-4(2):100–107, July 1968.

[12] Monika R. Henzinger, Philip Klein, Satish Rao, and Sairam Subramanian. Faster shortest-path algorithms for planar graphs. *Journal of Computer and System Sciences*, 55(1):3–23, 1997. (announced at STOC-94).

[13] Hermann Kaindl and Gerhard Kainz. Bidirectional heuristic search reconsidered. *Journal of Artificial Intelligence Research*, 7:283–317, 1997.

[14] George Karypis and Vipin Kumar. Multilevel k-way partitioning scheme for irregular graphs. *Journal of Parallel and Distributed Computing*, 48(1):96–129, 1998.

[15] S. Knopp, P. Sanders, D. Schultes, F. Schulz, and D. Wagner. Computing many-to-many shortest paths using highway hierarchies. manuscript in preparation, `http://algo2.iti.uka.de/schultes/hwy/`, 2006.

[16] Ekkehard Köhler, Rolf H. Möhring, and Heiko Schilling. Acceleration of shortest path and constrained shortest path computation. In *Proceedings of the 4th International Workshop on Experimental and Efficient Algorithms (WEA-05)*, volume 3503 of *LNCS*, pages 126–138. Springer, 2005.

[17] Ulrich Lauther. An extremely fast, exact algorithm for finding shortest paths in static networks with geographical backgrounds. In *Geoinformation und Mobilität - Von der Forschung zur praktischen Anwendung*, volume 22 of *IfGIprints*, 2004.

[18] Jens Maue. A goal-directed shortest path algorithm using precomputed cluster distances. Diploma thesis, Saarland University, Saarbrücken, 2006.

[19] Jens Maue, Peter Sanders, and Domagoj Matijevic. Goal directed shortest path queries using precomputed cluster distances. In *Proceedings of the 5th International Workshop on Experimental Algorithms (WEA-06)*, volume 4007 of *LNCS*, pages 316–327. Springer, 2006.

[20] Kurt Mehlhorn and Stefan Näher. *LEDA - A Platform for Combinatorial and Geometric Computing*. Cambridge University Press, Cambridge, UK, 1999.

[21] Metis: A software package for partitioning unstructured graphs, partitioning meshes, and computing fill-reducing orderings of sparse matrices. `http://glaros.dtc.umn.edu/gkhome/views/metis`, 1995.

[22] Ulrich Meyer. Average-case complexity of single-source shortest-path algorithms: lower and upper bounds. *Journal of Algorithms*, 48:91–134, 2003. (announced at SODA-01).

[23] Rolf H. Möhring, Heiko Schilling, Birk Schütz, Dorothea Wagner, and Thomas Willhalm. Partitioning graphs to speed up dijkstra's algorithm. *ACM Journal of Experimental Algorithms*, 11, 2007.

[24] Ira Pohl. Bi-directional search. In *Machine Intelligence*, volume 6, pages 127–140. Edinburgh University Press, 1971.

[25] Peter Sanders and Dominik Schultes. Highway hierarchies hasten exact shortest path queries. In *Proceedings of the 13th Annual European Symposium on Algorithms (ESA-05)*, volume 3669 of *LNCS*, pages 568–579. Springer, 2005.

[26] Mikkel Thorup. Undirected single-source shortest paths with positive integer weights in linear time. *Journal of the ACM*, 46(3):362–394, 1999. (announced at FOCS-97).

[27] Mikkel Thorup. Integer priority queues with decrease key in constant time and the single source shortest paths problem. *Journal of Computer and System Sciences*, 69(3):330–353, 2004. (announced at STOC-03).

[28] U.S. Census Bureau, Washington, DC. UA Census 2000 TIGER/Line Files. `http://www.census.gov/geo/www/tiger/tigerua/ua_tgr2k.html`, 2002.

[29] Dorothea Wagner and Thomas Willhalm. Geometric speed-up techniques for finding shortest paths in large sparse graphs. In *Proceedings of the 11th Annual European Symposium on Algorithms (ESA-03)*, volume 2832 of *LNCS*, pages 776–787. Springer, 2003.