Goal Directed Shortest Path Queries Using Precomputed Cluster Distances*

Jens Maue[1], Peter Sanders[2], and Domagoj Matijevic[1]

[1] Max-Planck-Institut für Informatik, 66123 Saarbrücken, Germany,
[jensmaue,dmatijev]@mpi-inf.mpg.de
[2] Universität Karlsruhe, 76128 Karlsruhe, Germany, sanders@ira.uka.de

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.

1 Introduction

Computing shortest paths in graphs (networks) with nonnegative edge weights is a classical problem of computer science. From a worst case perspective, the problem has largely been solved by Dijkstra in 1959 [1] who gave an algorithm that finds all shortest paths from a starting node s using at most $m+n$ priority queue operations for a graph $G=(V,E)$ with n nodes and m edges.

However, motivated by important applications (e.g., in transportation networks), there has recently been considerable interest in the problem of accelerating *shortest path queries*, i.e., the problem to find a shortest path between a source node s and a target node t. In this case, Dijkstra's algorithm can stop as soon as the shortest path to t is found. Furthermore, if the underlying graph does not change too often, it is possible to store some precomputed information that helps to accelerate the queries.

An extreme way to accelerate queries is to precompute *all* shortest path distances $d(s,t)$. However, this is impractical for large graphs since it requires quadratic space and preprocessing time. We explore the question whether it is possible to speed up queries by precomputing and storing only *some* shortest path distances.

Our approach, **P**recomputing **C**luster **D**istances (PCD), is very simple. We partition the graph into k disjoint clusters $\mathcal{V} = V_1 \dot\cup \cdots \dot\cup V_k$ and store the pair of starting and end point as well as the length of the shortest connection between each pair of clusters. This information needs space $\mathcal{O}(k^2)$ and can easily be computed using k executions of Dijkstra's algorithm. Refer to Section 3 for details. In Section 4 we explain how this information can be used to compute

* Partially supported by DFG grant SA 933/1-2.

upper and lower bounds for $d(s, t)$ at query time. These bounds can be used to prune the search. Section 5 completes the description of our algorithmic approach by presenting some fast partitioning heuristics. Most of them require no geometric information.

We then evaluate PCD in Section 6 using large, real-world road networks as they are used in car navigation systems. It is both geometrically plausible, and supported by our experiments that the speedup compared to Dijkstra's basic algorithm scales with $\sqrt{k}$, i.e., we get a flexible tradeoff between space consumption and query time that already yields useful acceleration for very small space consumption. To the best of our knowledge this is the first sublinear space acceleration technique that yields speedups $\gg 2$. Perhaps the most interesting application of PCD would be a combination with previous techniques that use linear space and deliver very high speedups but have no sense of goal direction [2–4]. Section 7 explores further future perspectives on PCD.

2 Related Work

Like most approaches to shortest paths with nonnegative edge weights, our method is based on Dijkstra's algorithm [1]. This algorithm maintains *tentative distances* $d[v]$ that are initially ∞ for *unreached nodes* and 0 for the source node s. A priority queue stores *reached* nodes ($d[v] < \infty$) using $d[v]$ as the priority. In each iteration, the algorithm removes the closest node u, *settles* it because $d[u]$ now is the shortest path distance $d(s, u)$, and *relaxes* the edges leaving u — if $d[u]+c((u, v)) < d[v]$, then $d[v]$ is decreased to $d[u]+c((u, v))$. When Dijkstra's algorithm is used for s–t shortest path queries, it can stop as soon as t is settled. When we talk about *speedup*, we mean the ratio between the complexity of this algorithm and the complexity of the accelerated algorithm.

A classical technique that gives a speedup of around two for road networks is *bidirectional search* which simultaneously searches forward from s and backwards from t until the search frontiers meet. Our implementation optionally combines PCD with bidirectional search.

Another classical approach is goal direction via *A* search*: a lower bound $\underline{d}(v, t)$ is used to direct the search towards the goal. This method can be interpreted as defining *vertex potentials* $p(v) := \underline{d}(v, t)$ and corresponding *reduced edge weights* $c^*((u, v)) := c((u, v)) + p(v) - p(u)$. Originally, lower bounds were based on the Euclidean distance from u to t and the "fastest street" in the network. Besides requiring geometric information, this very conservative bound is not very effective when searching *fastest* routes in road networks. In this respect, a landmark result was recently obtained by Goldberg and Harrelson [5, 6]. *Landmark A** uses precomputed distances to k *landmarks* to obtain lower bounds. Already around $k = 16$ carefully selected landmarks are reported to yield speedups around 70. Combined with a sophisticated implementation of *reach based routing* [2] this method currently yields the fastest query times for large road networks [4]. The main drawback of landmarks is the space consumption for storing $\Theta(kn)$ distances. This is where PCD comes into play, which already yields a very strong

sense of direction using much less space than landmarks. Still, the story is a bit more complicated than it sounds. We first considered PCD in March 2004 during a workshop on shortest paths in Karlsruhe. However, it turned out the lower bounds obtainable from PCD are *not* usable for A* search because they can lead to negative reduced edge weights.[3] Apparently, Goldberg et al. independently made similar observations [7]. The present paper describes a solution: use PCD to obtain *both* upper and lower bounds to prune search. The basic idea for PCD was presented in a Dagstuhl Workshop [8] (slides only).

There are several other speedup techniques that are based on partitioning the network into k clusters [9–11]. However, the preprocessing time required by these methods not only depends on k and the network size but also on the number B of border nodes, i.e., the number of nodes that have nodes from other clusters as neighbors. Furthermore, all of these methods need more space than PCD. Table 1 summarizes the tradeoff between space, preprocessing time and query time of these methods. Note that usually there is no worst case bound on the query time known. The given functions make additional assumptions that seem to be true for road network.

Table 1. Tradeoff between space, preprocessing time and query time depending on the choice of the parameter k for different speedup techniques, k is the number of clusters except for the landmark method. We assume road networks with n nodes and $\Theta(n)$ edges. $D(n)$ is the execution time of Dijkstra's algorithm, B is the number of border nodes. Preprocessing time does not include the time for partitioning the graph. (For the landmark method this is currently dominating the preprocessing time.)

Method		space	preprocessing	query
Edge Flags	[10, 9]	$\Theta(n \cdot k)$ Bits	$\Theta(B \cdot D(n))$	? ($\approx D(n)/2000@k = 128$)
Separator Hierarchy	[11]	$\Omega\left(B^2/k\right)$	$\Theta(B \cdot D(n/k))$	$\Omega\left(D(n/k) + B^2/k\right)$
Landmarks	[5]	$\Theta(n \cdot k)$	$\Theta(k \cdot D(n))$	? ($\approx D(n)/70@k = 16$)
PCD		$\Theta\left(k^2 + B\right)$	$\Theta(k \cdot D(n))$	$\Omega\left(D(n/\sqrt{k})\right)$

An interesting way to classify speedup techniques is to look at two major ways to prune the search space of Dijkstra's algorithm. A* search and PCD direct the search towards the goal. Other algorithms skip nodes or edges that are only relevant for short distance paths. In particular, reach based routing [2, 4] and highway hierarchies [3] achieve very high speedups without any sense of goal direction. Other techniques like edge flags [10, 9], geometric containers [12], and to some extent the landmark method show both effects. Therefore, we expect a major future application of PCD to augment highway hierarchies and reach based routing with a space efficient way to achieve goal direction.

[3] This was pointed out by Rolf Möhring.

3 Preprocessing

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

$$d(V_i, V_j) := \min_{s \in V_i, t \in V_j} d(s, t) \tag{1}$$

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'. Fig. 1 illustrates this approach.

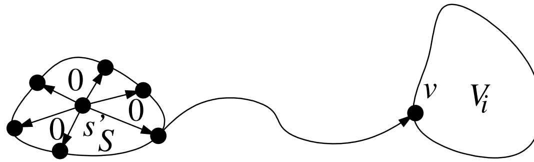

Fig. 1. Preprocessing connections from cluster S.

The following simple lemma shows that this suffices to find all connections from S to other clusters.

Lemma 1. $d(S, V_i) = \min_{v \in V_i} d(s', v)$.

The proof is almost obvious. We include it nevertheless since several other speedup techniques require shortest path computations from all *border nodes* of all partitions.

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 $v_i \in V_i$ and an end point $v_j \in V_j$ such that $d(v_i, v_j) = d(V_i, V_j)$.

4 Queries

We describe the query algorithm for bidirectional search between a source s and a target t. 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. Let S and T denote the clusters containing s and t respectively. The search works in two phases.

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 $d(s,t)$, and computing lower bounds $\underline{d}(s,w,t)$ for the length of any path from s via w to t. The search is pruned using the observation that the edges out of w need not be considered if $\underline{d}(s,w,t) > \hat{d}(s,t)$. Phase two ends 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 allow the proper initialization for the next query. This can be done efficiently by maintaining a stack of all nodes ever reached during the search. It remains to explain how $\hat{d}(s,t)$ and $\underline{d}(s,w,t)$ are computed.

The upper bound is updated whenever a shortest path to a node $u \in U$ is found such that u is the starting point of the shortest connection between clusters U and T. Let t_{UT} denote the stored end point of the precomputed shortest connection from U to T. Then we have

$$d(s,t) \leq d(s,u) + \underbrace{d(u, t_{UT})}_{=d(U,T)} + d(t_{UT}, t) \quad . \tag{2}$$

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 (2) is defined if $d(t_{UT}, t)$ is known, i.e. if t_{UT} has already been found by the backward search. Otherwise, we use an upper bound of the diameter of T instead of $d(t_{UT}, t)$ (see Section 5). $\hat{d}(s,t)$ is the smallest of the bounds from Equation (2) encountered during forward or backward search. The following lemma establishes the lower bound.

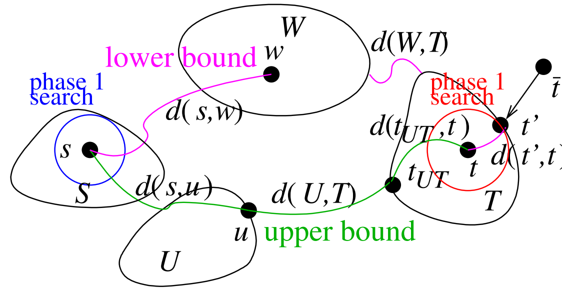

Fig. 2. Constituents of upper and lower bounds for $d(s,t)$.

Lemma 2. *Consider any node $w \notin T$. Let W denote the cluster containing w, and let* $\text{Border}(T) := \{t' \in T : \exists \bar{t} \notin T : (\bar{t}, t') \in E\}$ *denote the* border *of T, then any shortest path from s via w to t has a length of at least*

$$\underline{d}(s,w,t) := d(s,w) + d(W,T) + \min_{t' \in \text{Border}(T)} d(t',t) \ . \tag{3}$$

Proof. We show that $\underline{d}(s,w,t) \leq d(s,w) + d(w,t)$ or, equivalently, $d(W,T) + \min_{t' \in \text{Border}(T)} d(t',t) \leq d(w,t)$. Consider a shortest path $P = (w, \ldots, t'', \ldots t)$ from w to t where t'' denotes the first node on this path that is in cluster T. We have $d(w,t) = d(w,t'') + d(t'',t)$. Since $(w, \ldots, t'')$ is a connection from W to T we have $d(w,t'') \geq d(W,T)$. Furthermore, since t'' is a border node of T, we have $d(t'',t) \geq \min_{t' \in \text{Border}(T)} d(t',t)$. □

$\underline{d}(s,w,t)$ can be computed efficiently as $d(s,w)$ has been found by forward search, W can be found by storing a cluster identifier with each node, $d(W,T)$ has been precomputed, and $\min_{t' \in T} d(t',t)$ has been determined by the end of the first phase. Fig. 2 depicts the situation for computing upper and lower bounds.

Space Efficient Implementation

The algorithm described above is straight forward to implement using space $\mathcal{O}(k^2 + n)$. This can be reduced to $\mathcal{O}(k^2 + B)$ where B is the number of *border nodes* that have neighbors in other cluster. 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 all B border nodes in a hash table.

5 Partitioning

For any set $C = \{c_1, \ldots, c_k\} \subset V$ of k distinct *centers*, assigning each node $v \in V$ to the center closest to it results in a *k-center clustering*. Here, the *radius* $r(C_i)$ of a cluster C_i denotes the distance from its center c_i to the furthest member.[4] 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 C_i is deleted by removing the corresponding center c_i from C' and reassigning each member of C_i to the center now closest to it. 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

[4] Note that for undirected graphs, $2\,r(C_i)$ is an upper bound of the diameter of C_i since $d(u,v) \leq d(u,c_i) + d(c_i,v) \leq 2\,r(C_i)$ for any $u, v \in C_i$. This bound can be used in the query as shown in Section 4. For directed graphs we can use $r(C_i) + \max_{c \in C_i} d(c, c_i)$ for the same purpose.

Section 6 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. It can be shown that partitioning using the MinSize heuristic searches $\mathcal{O}(n \log \frac{k'}{k})$ nodes using Dijkstra's algorithm and hence has negligible cost compared to computing cluster distances which requires searching $\mathcal{O}(nk)$ nodes.[5] The radius of a cluster affects the lower bounds of its members, and it seems that a good partitioning for PCD has clusters of similar size and a low average radius. Oversampling indeed keeps a low average radius since deleted clusters tend to be distributed to neighbors of lower radius. However, 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*, in which the single radii are weighted with their clusters' sizes.

Our k-center heuristics are compared with a simple partitioning based on a rectangular grid and with Metis [13]. Metis was originally intended for parallel processing where partitions should have close to equal size and small boundaries in order to reduce communication volume.

6 Experiments

The PCD algorithms were implemented in C++ using the static graph data structure from the C++ library LEDA 5.1 [14] and compiled with the GNU C++ compiler 3.4 using optimization level `-O3`. All tests were performed on a 2.4 GHz AMD opteron with 8 GB of main memory running Linux (kernel 2.6.11). We use the same input instances as in [3]—industrial data for the road network of Western Europe and freely available data for the US [15]. Table 2 gives more details. In order to make experiments with a wide range of parameter choices, we used subgraphs of these inputs. Unless otherwise noted, the experiments make the following common assumptions: we use undirected graphs in order to be able to use simple implementations of the partitioning heuristics. (Our PCD implementation works for general directed graphs.) Edge weights are estimated travel

[5] Throughout this paper $\log x$ stands for $\log_2 x$.

Table 2. The graph instances used for experiments.

Instance	n	m	Description
`DEU`	4 375 849	5 483 579	Germany
`SCA`	2 453 610	2 731 129	Sweden & Norway
`IBE`	872 083	1 177 734	Spain & Portugal
`SUI`	630 962	771 694	Switzerland
`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)
`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)

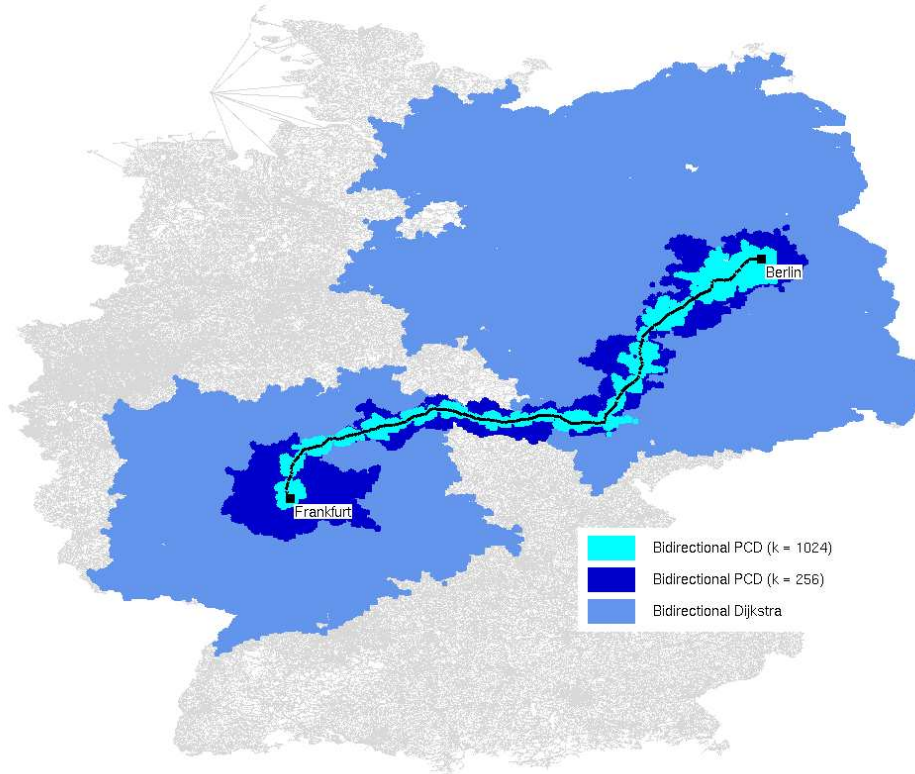

Fig. 3. The search space for a sample query from Frankfurt to Berlin.

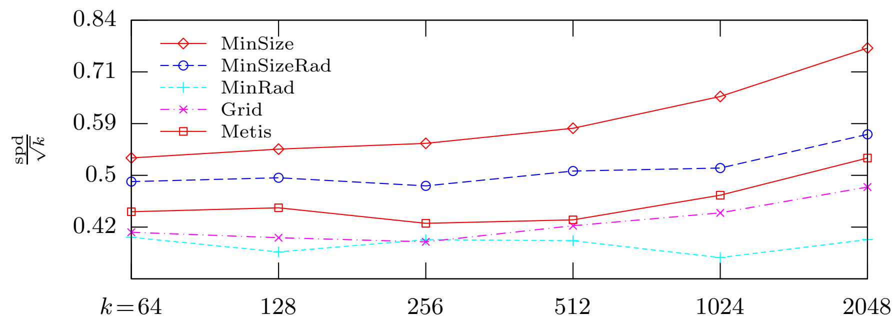

Fig. 4. Scaled speedups for bidirectional PCD depending on the method of clustering.

times. The default instance is the road network of Germany (`DEU`). Partitioning is done using $k \log k$-oversampling with the MinSize heuristic. The speedup is the ratio between the number of nodes settled by Dijkstra's unidirectional algorithm and by the accelerated algorithm. The given values for queries are averages over 1000 random query pairs.

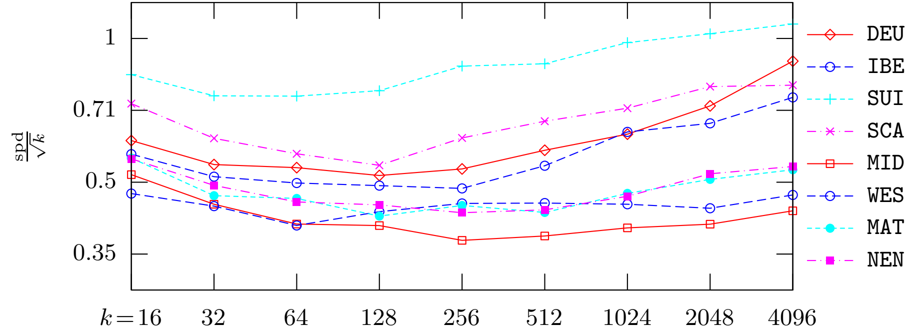

Fig. 5. Scaled speedups for bidirectional PCD and different inputs.

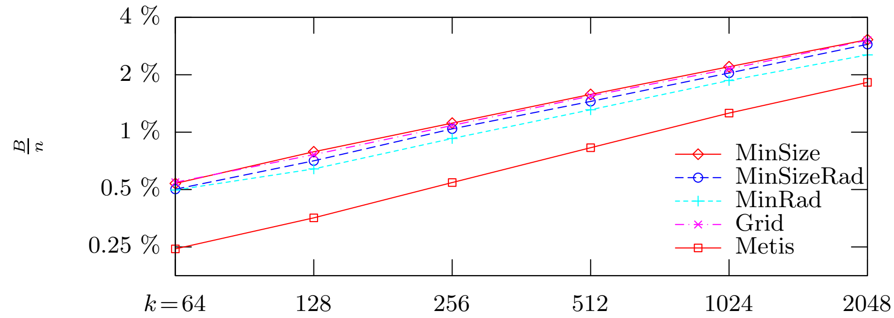

Fig. 6. Relative number of border nodes B for `DEU` depending on the clustering method. The k-center heuristics all use $k \log k$-oversampling.

Fig. 3 gives an example for the shape of the search space explored by PCD. As to be expected, the search space is a kind of halo around the shortest path that gets narrower when k is increased. A somewhat optimistic approximation of the observed behavior is that the clusters near the shortest path are explored. The shortest path will intersect $\mathcal{O}(\sqrt{k})$ clusters of size $\mathcal{O}(n/k)$ on the average, i.e., the intersected clusters contain $\mathcal{O}(n/\sqrt{k})$ nodes. Since Dijkstra's algorithm visits $\Theta(n)$ nodes on the average, we expect a speedup of $\mathcal{O}(\sqrt{k})$.

This hypothesis is verified in Fig. 4, which compares the different partitioning methods from Section 5. Scaled by $\sqrt{k}$, the speedups describe fairly flat lines as our hypothesis suggests. The MinSize heuristic yields the highest speedups, while the other heuristics perform worse though they also yield fairly small average radii as mentioned before. Since MinRad keeps deleting clusters in urban areas, it ends up with clusters of similar radii but differing sizes, wheras MinSize deletes clusters regardless of their size yielding a good ratio of radius and size. Interestingly, for the minimum size rule the speedups appear to scale even better than $\Theta(\sqrt{k})$. This observation is confirmed in Fig. 5 for further instances.

As expected, Metis finds clusters with smaller borders as can be seen in Fig. 6. However, since the percentage of border nodes is very small even for oversampling, this appears to be less relevant.

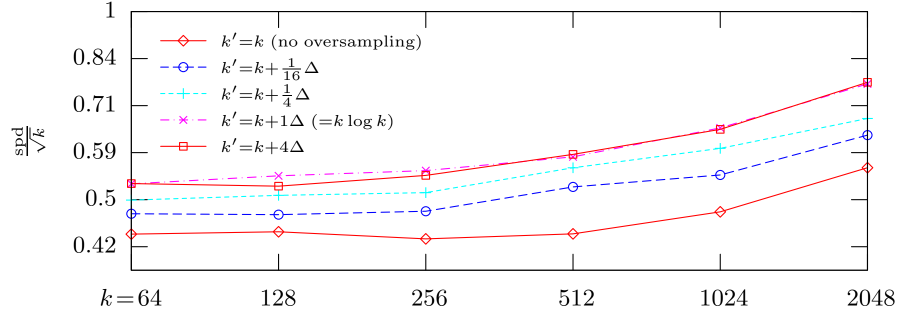

Fig. 7. Speedups for bidirectional PCD with k'-oversampling using different values for k', tested on `DEU`. $\Delta = k \log k - k$ denotes the difference between $k \log k$ and k.

Oversampling using the MinSize deletion heuristic is tested for several values of k' in Fig. 7. Starting from $k' = k$, which means just choosing k centers randomly, increasing k' up to $k \log k$ increases the speedup significantly. However, no considerable further improvement is visible for $k' > k \log k$.

Table 3 summarizes our results for several instances and different numbers of clusters k calculated by $k \log k$-oversampling.The highest speedup in our tests is 114.9, while speedups of more than 30 are still achieved while the number of cluster pairs k^2 remains significantly below n and the total number of border nodes B is negligible. The speedups for the US instances are smaller probably because the available data provides less information on speed differences of roads, while using travel distances rather than travel times seems to yield even smaller speedups. The reason is that edges off the fastest path often have high travel time values compared to edges on the path, so that pruning happens earlier.

Table 3. Performance of PCD for several graph instances and selected values of k. prep.= preprocessing time. B = total number of border nodes. spd = speedup. settled = number of settled nodes. t = query time.

graph	k	$\frac{B+k^2}{n}$	prep.	PCD unidirectional				PCD bidirectional			
			[min]	t[ms]	spd	settled	spd	t[ms]	spd	settled	spd
`DEU`	2^4	< 0.01	2.6	2491	2.2	1 171 110	2.2	2114	2.5	1 028 720	2.4
	2^6	0.01	11.1	1410	3.6	749 870	3.3	971	5.2	553 863	4.3
	2^8	0.03	35.0	677	7.7	443 832	5.9	422	12.3	295 525	8.5
	2^{10}	0.26	123.0	256	21.2	199 663	13.2	157	35.0	127 604	20.2
	2^{12}	3.88	558.2	110	70.5	86 401	37.1	62	114.9	50 417	57.4
`SCA`	2^{10}	0.44	60.7	173	21.3	136 365	14.2	81	36.5	70 766	22.9
`IBE`	2^{10}	1.25	11.7	43	16.5	51 716	13.7	20	24.7	26 591	20.4
`SUI`	2^{10}	1.71	11.1	20	20.8	25 070	19.1	9	31.1	12 848	31.4
`MID`	2^{10}	0.22	89.2	287	15.4	326 112	10.3	223	18.5	242 153	12.8
`WES`	2^{10}	0.25	80.8	238	15.2	227 768	11.3	169	19.0	159 409	14.4
`MAT`	2^{10}	0.50	35.5	101	17.1	114 702	12.0	76	21.1	83 577	15.2
`NEN`	2^{10}	1.21	11.0	39	15.1	49 259	11.6	28	19.3	34 625	15.0

The speedups in terms of query time are higher than those in terms of settled nodes. The main reason for this lies in the size of the priority queue in PCD, which affects the average time for queue operations: after a small ball around the source is searched, most of the nodes on its boundary are pruned, and the search frontier turns into a small corridor around the shortest path. Then, the queue holds a number of nodes which remains roughly constant until finishing. The queue size corresponds to the width of the corridor, so the average queue size is in $\mathcal{O}(\sqrt{\frac{n}{k}})$, while in Dijkstra's algorithm the queue keeps growing and holds $\mathcal{O}(n)$ nodes on the average. Since the average time of an operation is logarithmic in the size and the number of operations is linear in the number of settled nodes, the relation between the speedup in terms of query time and that in terms of settled nodes is roughly $\frac{\log n}{\log \frac{n}{k}}$.

7 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 this with speedup techniques that have no sense of goal direction [2–4]. There are good reasons to believe that one would get a better tradeoff between speedup and space consumption than any previous method.

As a standalone method, PCD is interesting because its unidirectional variant also works for networks with time dependent edge weights such as public transportation networks or road networks with information when roads are likely to be congested: simply use an optimistic estimate for lower bounds and a pessimistic estimate for the upper bounds. Most other speedup techniques do not have such an obvious generalization.

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; perhaps, these might be two of the main parameters for an easily computable objective function. In the literature there is a lot of work on approximation algorithms for various k-center problems. It might be interesting to adapt some of the proposed algorithms to our situation.

References

1. Dijkstra, E.W.: A note on two problems in connexion with graphs. Numerische Mathematik **1** (1959) 269–271
2. Gutman, R.: Reach-based routing: A new approach to shortest path algorithms optimized for road networks. In: 6th Workshop on Algorithm Engineering and Experiments. (2004)
3. Sanders, P., Schultes, D.: Highway hierarchies hasten exact shortest path queries. In: 13th European Symposium on Algorithms. Volume 3669 of LNCS., Springer (2005) 568–579
4. Goldberg, A., Kaplan, H., Werneck, R.: Reach for A^*: Efficient point-to-point shortest path algorithms. In: Workshop on Algorithm Engineering & Experiments, Miami (2006)
5. Goldberg, A.V., Harrelson, C.: Computing the shortest path: A^* meets graph theory. In: 16th ACM-SIAM Symposium on Discrete Algorithms. (2005) 156–165
6. Goldberg, A.V., Werneck, R.F.: An efficient external memory shortest path algorithm. In: Workshop on Algorithm Engineering & Experiments. (2005) 26–40
7. Goldberg, A.: personal communication. (2005)
8. Sanders, P.: Speeding up shortest path queries using a sample of precomputed distances. Workshop on Algorithmic Methods for Railway Optimization, Dagstuhl, June 2004, slides at `http://www.dagstuhl.de/04261/` (2004)
9. Möhring, R.H., Schilling, H., Schütz, B., Wagner, D., Willhalm, T.: Partitioning graphs to speed up Dijkstra's algorithm. In: 4th International Workshop on Efficient and Experimental Algorithms. (2005)
10. Lauther, U.: An extremely fast, exact algorithm for finding shortest paths in static networks with geographical background. In: Münster GI-Days. (2004)
11. Schulz, F., Wagner, D., Zaroliagis, C.D.: Using multi-level graphs for timetable information. In: 4th Workshop on Algorithm Engineering and Experiments. Volume 2409 of LNCS., Springer (2002) 43–59
12. Wagner, D., Willhalm, T.: Geometric speed-up techniques for finding shortest paths in large sparse graphs. In: 11th European Symposium on Algorithms. Volume 2832 of LNCS., Springer (2003) 776–787
13. Karypis, G., Kumar, V.: Metis: A software package for partitioning unstructured graphs, partitioning meshes, and computing fill-reducing orderings of sparse matrices. `http://www-users.cs.umn.edu/~karypis/metis/` (1995)
14. Mehlhorn, K., Näher, S.: The LEDA Platform of Combinatorial and Geometric Computing. Cambridge University Press (1999)
15. U.S. Census Bureau: UA Census 2000 TIGER/Line files. `http://www.census.gov/geo/www/tiger/tigerua/ua_tgr2k.html` (2002)