The VLDB Journal
DOI 10.1007/s00778-015-0392-3

SPECIAL ISSUE PAPER

On k-Path Covers and their applications

Stefan Funke[1] · **André Nusser**[1] · **Sabine Storandt**[2]

Received: 5 December 2014 / Revised: 20 April 2015 / Accepted: 20 June 2015

Abstract For a directed graph G with vertex set V, we call a subset $C \subseteq V$ a *k-(All-)Path Cover* if C contains a node from *any* simple path in G consisting of k nodes. This paper considers the problem of constructing small k-Path Covers in the context of road networks with millions of nodes and edges. In many application scenarios, the set C and its induced overlay graph constitute a very compact synopsis of G, which is the basis for the currently fastest data structure for personalized shortest path queries, visually pleasing overlays of subsampled paths, and efficient reporting, retrieval and aggregation of associated data in spatial network databases. Apart from a theoretic investigation of the problem, we provide efficient algorithms that produce very small k-Path Covers for large real-world road networks (with a posteriori guarantees via instance-based lower bounds). We also apply our algorithms to other (social, collaboration, web, etc.) networks and can improve in several instances upon previous approaches.

Keywords Path cover · Graph compression · VC-dimension · Pruning algorithm · Personalized route planning · Spatial network database

✉ Sabine Storandt
storandt@cs.uni-freiburg.de

Stefan Funke
funke@fmi.uni-stuttgart.de

André Nusser
nusser@fmi.uni-stuttgart.de

[1] Universität Stuttgart, FMI, Universitätsstr. 38, 70569 Stuttgart, Germany

[2] Universität Freiburg, Georges-Köhler-Allee 51, 79110 Freiburg im Breisgau, Germany

1 Introduction

The massive acquisition of geospatial data in the course of collaborative projects like OpenStreetMap (OSM)[1] or by companies like Google or TomTom has led to a dramatic growth of data to be handled in spatial network databases (SNDB). For example, the OSM 'world graph' at the beginning of 2007 contained >30 million nodes, whereas in 2013 this number has grown to more than two billion nodes. A limit to this growth is nowhere to be seen due to the demand for a more and more accurate and detailed representation of our environment. SNDBs manage geographic entities located in an underlying *road network* supporting efficient data retrieval operations, in particular taking into account connectivity properties of the road network. Google/Bing/ Yahoo Maps are all incarnations of SNDBs. Let us look at a few applications for SNDBs which can benefit from small k-Path Covers.

Application 1—facility location If one wants to place certain facilities in a road network, like gas stations or traffic signs, the goal is to cover every path of a certain length (so vehicles do not run out of fuel, or people do not get lost). As placing facilities generates costs, one aims at finding a placement which minimizes the number of necessary facilities. This problem has a one-to-one correspondence to k-Path Cover, with k capturing the notion of the path length and the resulting cover $C \subseteq V$ defining the facility positions. See Fig. 1 for an example.

Application 2—data aggregation Assume we have computed or decided on some (not necessarily shortest or quickest) route π for a weekend trip and are interested in shopping or refueling opportunities along the route. If we had a data structure $\mathcal{D}$ which can retrieve for any node $v \in V$ in the network nearby points of interest like shopping malls or gas

[1] http://www.openstreetmap.org.

Published online: 14 July 2015

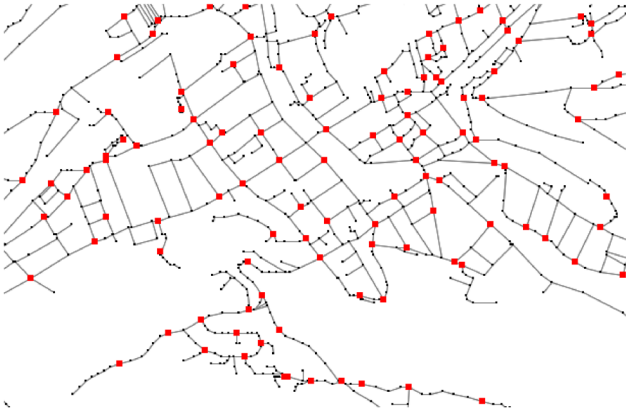

Fig. 1 Valid set of facility locations (*red boxes*) to cover the graph using $k = 9$

Fig. 2 Route with nearby gas stations and forest areas. *Red nodes* indicate a subsampling resulting from a global cover computation. Querying only the *red points* instead of all points to identify gas stations or path sections through the forest is much faster, but still sufficiently accurate

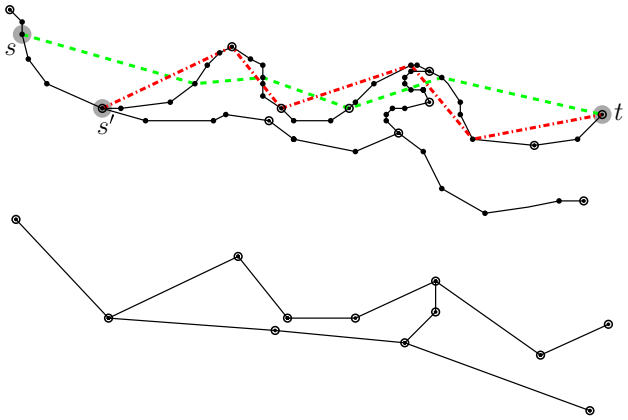

Fig. 3 Small hiking map. In the upper image, the *dashed red* and *green line* indicate two hiking routes ($s \to t$ and $s' \to t$) simplified greedily using $k = 6$. While $s' \to t$ is a subroute of $s \to t$, the two simplified routes do not resemble each other at all. The *circled* nodes are a feasible global cover for $k = 6$, resulting in a nice simplification of the map presented in the lower image, also allowing for consistent downsampling

stations, we could query $\mathcal{D}$ with every node on π to get the desired answer. With π possibly consisting of hundreds or thousands of nodes, it might be more efficient (but sufficiently accurate) to query $\mathcal{D}$ with only every k-th node on π (e.g., $k = 10$). Still, this requires having constructed $\mathcal{D}$ for *all* nodes of the network. An elegant solution approach is to identify a small subset $C \subseteq V$ such that for any path consisting of k nodes, at least one node is contained in C—and construct $\mathcal{D}$ for C only. Now for any given path π, we can query at least every k-th node using $\mathcal{D}$. Similarly, if the goal is to acquire statistics along paths (like e.g., the percentage of forest/desert coverage along routes throughout the USA), it is more efficient to aggregate that data at the nodes determined by C in terms of both space requirement and running time for a single query. See Fig. 2 for a visualization.

Application 3—map simplification In another scenario, a web portal dealing with scenic hiking trips sends several suggested hiking routes to the clients for visualization in a browser. In particular in a zoomed out view, it is a waste of bandwidth to transmit every single node of each route across the internet; subsampling the paths, e.g., every k-th node, is the preferred method. If routes overlap, though, the chosen subsampling for the routes should be consistent to avoid visual artefacts. Again, if we could determine a subsample C of V which guarantees that in any route at least every k-th node is present, a more pleasing visual appearance can be achieved when rendering an overlay of the subsampled routes; see Fig. 3.

Application 4—personalized route planning Next-generation route planners allow for personalized route planning queries where each user has an individual profile depending on his personal preferences and driving styles. For example, the typical driving speed varies (speeder or slow driver), people like to trade gas price against travel time, and different vehicles exhibit different turn costs. So a query might consist not only of source and destination but also of a set of parameters which determine under which metric the optimal path in the road network is to be computed. If we construct an overlay graph on a small subsample C of V where C hits *any* k-path in the road network, we have a metric-independent compressed graph at hand. For any kind of input parameters specified by a user, we can compute the respective edge costs in this graph on demand and find the optimal path by a search in the overlay graph considerably faster than a search in the original graph.

We want to emphasize that for all these applications, it is crucial that the Path Cover C hits *any* path consisting of k nodes, not only shortest or quickest paths under some fixed metric.

In general, computing a compact graph synopsis is of interest not only in the context of SNDBs but for many application domains where large-scale networks occur, e.g., social and collaboration networks or web graphs.

1.1 Related work

Tao et al. in [18] considered the problem of computing a *k-Shortest*-Path Cover, that is, a set of nodes $C \subseteq V$ such that

C contains at least one node from every *shortest* path (under some *fixed* metric) consisting of k nodes. More concretely, they could, for example, construct a set C which was only 15 % of the size of V for $k = 16$ and the US road network. As one application example, they showed how to use such a small C to accelerate shortest path queries (for a *fixed* metric) via the overlay graph induced by C (and in combination with additional speedup techniques like *reach* [13]). We want to note, though, that the achievable query times have meanwhile been superseded by current speedup techniques like *transit nodes* [3], contraction hierarchies (CH) [12] or *hub labels* [2] which answer queries faster by several orders of magnitude. A fundamental restriction of all these techniques (including [18]) is that they all rely on fixed edge weights for the preprocessing stage. If edge weights change (e.g., for a different user profile), the preprocessing has to be revised or even redone from scratch again, making all these approaches unsuitable for the use case where every query comes with a different user profile.

In [6] a three-phase approach was introduced to allow for customizable route planning (CRP). The first phase is a metric-independent graph preprocessing purely based on the topology of the graph. In the second phase, a specific metric is considered and the graph is customized accordingly. In the third phase, queries can be answered efficiently for the metric specified in the second phase. In fact, for their benchmark graph of Western Europe (around 18 million vertices), query times below 1 ms could be achieved. But as soon as the metric changes, phase two has to be redone. For the CRP framework, the metric customization takes several CPU core seconds (on a rather powerful machine). While this is fine for simulating traffic-dependent edge costs or when customizing for a specific *single* user, several seconds is too slow when every query demands its own metric (in fact running plain Dijkstra might then be faster). More recently, the CRP approach has been refined such that customization can be performed on graphics hardware [7]. Now customization times of a fraction of a second can be achieved—but only via heavy (CPU or GPU) parallelism. On a single CPU core, customization time is still more in the order of an ordinary Dijkstra run.

The focus of our work is the highly dynamic scenario where every single query comes with an individual metric. We also employ a metric-independent preprocessing first, but then merge the second and third phase such that metric changes due to different input parameters can be dealt with on query time. In [11] the authors adapt the CH preprocessing technique to the case where each edge e in the network has r associated costs $c_1(e), c_2(e), \ldots, c_r(e)$. A query consists of source, destination and nonnegative multipliers $\alpha_1, \alpha_2, \ldots, \alpha_r$; the data structure returns the shortest paths under edge costs $c(e) = \alpha_1 c_1(e) + \alpha_2 c_2(e) + \cdots + \alpha_r c_r(e)$. Experimental results proved the technique to yield high speedups for $r = 2$ and 3. Still, the speedup for $r = 3$ was already considerably lower than for $r = 2$, as with each additional metric the hierarchy of optimal paths (which is crucial for the approach) subsides. Moreover, it was shown that considering metrics which are orthogonal to each other (like low travel time and preferring quiet roads at the same time) is disadvantageous. For a larger number (e.g., $r > 4$) of dissimilar costs, this approach reaches its limits, especially as the preprocessing gets then very complicated, and the preprocessing time grows exponentially in r.

In [17] the problem of constructing a gate graph was introduced, which is closely related to the k-Shortest-Path Cover problem discussed in [18]. A gate graph is based on the selection of a subset of vertices (the so called minimum gate vertex set, or MGS for short) of the original graph, which fulfills the following property: For every pair of vertices v, w in the original graph, there exists a sequence of gate vertices $g_1, g_2, \ldots, g_r$ such that the path from v to w over these gate vertices yields a minimal hop distance between v and w; and neither two consecutive gate vertices, nor v and g_1 or g_r and w are more than ϵ hops apart for given parameter $\epsilon \in \mathbb{N}$. Then gate vertices with a hop distance of at most ϵ are connected with direct edges (augmented with suitable weights) to form the gate graph. The main application discussed in the paper is the simplification of undirected and unweighted graphs. In fact, a k-Shortest-Path Cover with $k = \epsilon - 1$ solves the gate vertex problem as already observed in [17]. Nevertheless, a new algorithm to find a concise gate vertex set was introduced which is based on reducing the problem to an instance of the Set Cover problem. It was shown that smaller gate vertex sets were achieved on social networks with the new algorithm compared to the method proposed in [18]. But the set cover-based approach is limited to rather small values of ϵ (evaluations up to $\epsilon = 6$ in the paper) and small networks due to the high time and space consumption of the used algorithms. Among applications for which their algorithms apply, the authors of [17] refer to the analysis social networks, collaboration networks and web graph excerpts. Several benchmark results for such nonroad network type instances are given in their paper.

1.2 Our results

The main contributions of our paper are the following:

- We generalize the work of [18] by considering *all paths* in the network instead of only *shortest paths* and devise efficient algorithms to compute small k-Path Covers. The resulting covers are even smaller than the covers reported in [18] but with the much stronger property of covering *all* k-paths instead of only shortest k-paths.
- As a by-product, we devise an algorithm that constructs *considerably smaller* sets C with the same property (covering *shortest* k-paths) as the ones derived in [18]; for

example, for the US road network and $k = 16$, we can find a set C which is only 5.8 % the size of $|V|$ (compared to 15 % reported in [18]).
- More on the theoretic side, we show that the problem of covering all k-Shortest-Paths can be approximated within a logarithmic factor of the optimal solution by using ϵ-net theory and proving a new result on the VC-dimension of directed shortest path systems.
- As a concrete application for our k-Path Covers, we devise the to our knowledge to-date fastest scheme to answer personalized route planning queries.

Apart from a more detailed exposition of the above, in this extended version of the original paper [10], we present several new results in the context of k-Path Covers:

- We investigate (theoretic) lower bounds on the complexity and approximation quality of our approach.
- For the k-Shortest-Path Cover problem, we develop an acceleration technique for our algorithm based on estimating upper bounds for shortest path distances. This new technique reduces the cover computation time significantly—especially for large values of k.
- We discuss in detail how to construct overlay graphs induced by k-Path Covers. For Shortest-Path Cover overlay graphs, we present a method which outperforms previous approaches (as developed in [18]) in both running time and solution quality.
- We investigate the performance of our algorithms on several nonroad network type problem instances, in particular social networks, collaboration networks and excerpts from the web graph. These networks differ significantly from road networks in terms of maximum/average node degree and diameter. We empirically prove that our algorithms perform well also in these settings and are able to tackle much large instances than considered in previous work.
- Furthermore, we added new experiments concerning nested cover construction and graph/path compression. We also refined our results on personalized route planning and considered more difficult instances (larger k-value for the biggest network) for this application than before.

1.3 Outline

After providing formal definitions in Sect. 2, we review in Sect. 3 theoretic results on the complexity of the k-Path Cover problem, and the k-Shortest-Path Cover problem, respectively. As k-Path Cover is APX-hard [4] and k-Shortest-Path Cover turns out to be APX-hard as well, we investigate approximation algorithms based on low VC-dimension of the underlying set system. In Sect. 4 we develop practical algorithms to construct k-All-Path Covers and k-Shortest-Path Covers as well as instance-based lower bounds. In Sect. 5 we investigate the efficient construction of the overlay graph induced by a Path Cover, which is important for several applications mentioned in the introduction. In fact, we need the overlay graph as an ingredient for personalized route planning. In Sect. 6 we describe in very detail how to preprocess the road network in order to enable real-time query answering for this application. We conclude with an experimental study described in Sect. 7, showing the practicability of our developed algorithms and in particular looking at the use case of personalized route planning. We also investigate the applicability of our algorithms for other network types besides road networks.

2 Formal definitions

To formalize our cover problems, we introduce the following notations: The input is a simple directed graph $G(V, E)$ (simple means G contains no self-loops and no multi-edges). Therefore, a path π in G can be uniquely represented as the list of its traversed vertices or edges. We will use both representations depending on what is more convenient. Throughout this paper, when using the term 'path' we always refer to simple paths, that is, no vertex appears more than once. Furthermore, we define the length $|\pi|$ of a path π to be the number of nodes it contains including the source and target vertex.

Our focus lies on determining a small subset of nodes such that *all* paths contained in G are subsampled sufficiently by those nodes. The density of the sampling can be chosen in an application-dependent manner via the input parameter k.

Definition 1 *(Minimum k-(All-)Path Cover)* Given a (directed) graph $G(V, E)$ and $k \in \mathbb{N}$, select a minimum subset of vertices $C \subseteq V$ such that for every simple path $\pi = v_1, \ldots, v_k$ in G we have $C \cap \pi \neq \emptyset$. We will refer to this problem as k-APC.

Of course, a cover for all paths also yields a feasible cover if we are only interested in subsampling *shortest* paths under a specific metric. Nevertheless, the problem of covering only shortest paths allows for specializations of the algorithms for the general case, which leads to even more compact cover sets C. Hence, we also define the cover problem for shortest paths.

Definition 2 *(Minimum k-Shortest-Path Cover)* Given a weighted (directed) graph $G(V, E, c)$, $c : E \to \mathbb{R}_0^+$ being the weights, and $k \in \mathbb{N}$, select a minimum subset of vertices $C \subseteq V$ such that for every shortest (according to c) path $\pi = v_1, \ldots, v_k$ in G we have $C \cap \pi \neq \emptyset$. We will refer to this problem in the following as k-SPC.

Note that in contrast to [18] our definition for k-SPC requires covering *all* shortest paths, which appears harder if several shortest paths between some pairs of nodes exist.

Our main application—the efficient computation of personalized shortest path queries—will take advantage of the fact that a k-APC can also be seen as a k-SPC for *all* possible metrics. More concretely, we are given r edge cost functions $c_1, c_2, \ldots, c_r : E \to \mathbb{R}_0^+$ and each query specifies—apart from source and target nodes $s, t \in V$—which costs are important and to what extent (which can be realized by choosing weighting factors $w_1, w_2, \ldots, w_r \in \mathbb{R}_0^+$). The goal is then to find the path π from s to t, which minimizes the aggregated weighted costs $\sum_{e\in\pi} \sum_{i=1}^{r} w_i \cdot c_i(e)$. To allow for efficient answering of such queries, we will first compute a concise k-APC $C \subseteq V$ in a preprocessing phase and then evaluate edge costs between nodes in C at query time according to the query weights.

3 Theoretic analysis

In this section, we give a brief overview of complexity results for k-APC and k-SPC and show that k-SPC is hard to approximate better than a factor of 2 in the general case. Then we derive a $\log |OPT|$- approximation for k-SPC based on VC-dimension analysis.

3.1 NP-hardness and approximation

In [17], the MGS problem was also analyzed theoretically. Here, the authors proved NP-hardness for a generalized problem version, namely, in terms of our setting, a variant, where only a prespecified subcollection of shortest paths of length k has to be hit. However, the complexity of hitting *all* shortest paths of length k remains unclear there, as hitting all paths is a special case of hitting a subcollection—and therefore might be easier to solve than the generalized version. A thorough theoretic analysis of k-APC was conducted in [4]. Via reduction from the VertexCover problem, the authors proved APX-hardness of k-APC (subsuming NP-hardness). In particular, they showed that for $k > 2$ an approximation better than 1.3606 in polynomial time demands P=NP. We remark that if the unique game conjecture [15] holds, their results even imply APX-hardness within a factor better than 2 (see [16]). Furthermore we would like to point out that this APX-hardness result carries over to k-SPC as discussed in more detail in the following Lemma.

Lemma 1 *k-SPC is hard to approximate better than a factor of 2 if the unique game conjecture holds.*

Proof In a VertexCover instance, one is given a graph $G(V, E)$ and the goal is to find a subset $C \subseteq V$ of minimum size, such that at least one endpoint of every edge is contained in C. To reduce the problem to k-SPC, we construct a graph $G'(V', E')$ with the following properties (similar to the approach in [4]):

- V' contains whole V and E' contains whole E.
- For every vertex in V, an auxiliary path gets attached to it which contains $\lfloor (k-1)/2 \rfloor$ vertices. These vertices and edges are added to V' and E'.
- All edges in E' are augmented with costs of 1.

The transformation clearly demands only polynomial time.

A VertexCover C in G is a k-SPC $C' = C$ in G', as the longest path without a node in C contains at most $1 + \lfloor (k-1)/2 \rfloor$ vertices and the term is smaller than k for any $k > 1$.

For a k-SPC C'in G', we first snap vertices in $V' \setminus V$ to the closest vertex in V according to the hop distance, providing us with a new set C for which we can guarantee $|C| \leq |C'|$. If this set C is not a VertexCover in G, there has to be some edge $\{v, w\} \in E$ and neither v, w nor one of the auxiliary paths attached to those vertices have a nonempty intersection with C. Let v' and w' be the vertices on those auxiliary paths with the largest distance to v and w, respectively, then $v', \ldots, v, w, \ldots, w'$ is the unique shortest path from v' to w'. As this path contains $2\lfloor (k-1)/2 \rfloor + 2 > k$ vertices, C has to contain at least one of its vertices to be a valid k-SPC. This contradicts the fact that $\{v, w\}$ is not hit by C. Hence, every edge is hit by C, and C is a valid VertexCover in G.

Considering both directions, we conclude that the solution sizes $|C|$ and $|C'|$ have to be equal. Therefore, we can solve the VertexCover problem via reduction to the k-SPC problem in a solution size preserving manner. Hence, all hardness results for VertexCover transfer to k-SPC, which in particular means that k-SPC is inapproximable with a factor better than 2 if the unique game conjecture holds. □

So approximation algorithms with a guarantee better than a factor of 2 seem to be out of reach for both k-APC and k-SPC. But as observed in [4] for the k-APC problem, there exists an algorithm that guarantees a k-approximation based on converting the problem to an instance of HittingSet (as defined for the sake of notation in the following).

Definition 3 *(HittingSet)* Given a set system (U, S) with U being a universe of elements and S a collection of subsets $S_i \subseteq U$. Find a minimum set $U^* \subseteq U$ such that $\forall S_i \in S : S_i \cap U^* \neq \emptyset$, i.e., every set in S is hit by U^*.

In fact, a primal-dual algorithm exists (the so called pricing method), which provides a k-approximation for HittingSet if all S_i have a size $\leq k$. In our setting, we interpret every path of length k as set of its contained vertices to construct the HittingSet instance. This construction also works for k-SPC when restricted to shortest paths. So for k-APC and

k-SPC every set has exactly k elements, immediately implying a valid k-approximation. Of course, the reduction to a HittingSet instance also provides us with a $\ln n + \Theta(1)$ approximation via the standard greedy algorithm (with $n = |V|$ denoting the number of nodes in the network). This approximation guarantee might be tighter in case of large k.

3.2 VC-dimension of directed shortest paths

To obtain an upper bound for the size of a k-SPC, the theory of ϵ-nets was applied in [18]. An ϵ-net for a set system (U, S) is a HittingSet for all elements in S which satisfy $|S_i| \geq \epsilon |U|$ for some $\epsilon \in (0, 1)$. As shown in [18], an ϵ-net with $\epsilon = k/n$ is a k-SPC for the set system where U equals the set V of nodes in G and S the set of all shortest paths. Applying the ϵ-net theorem [14], we can find a k-SPC of size $\mathcal{O}(d(n/k)\log(n/k))$ with d being the Vapnik–Chervonenkis(VC) dimension of the set system [19].

Definition 4 *(VC-Dimension)* The VC-dimension d of a set system (U, S) is defined as the size of the largest subset $U' \subseteq U$ that can be shattered. Here, a subset U' is called *shattered* if for any subset $A \subseteq U'$, there exists $B \in S$ with $U' \cap B = A$.

So in case S consists of a set of paths, the VC-dimension describes an exclusive upper bound on how often two paths can intersect in a noncontiguous manner. Obviously, if shortest paths are ambiguous in G, the VC-dimension depends on the length of the paths. But as already shown in [17], for every G there exists a system of unique shortest paths that can be investigated. The same effect can be achieved by symbolic perturbation of the edge weights. For *undirected* shortest paths, it was proven in [1] (in the context of analyzing shortest path speedup techniques) and [18] that the VC-dimension is at most 2. Hence, we can find a k-SPC containing no more than $\mathcal{O}(n/k \log(n/k))$ elements. It was remarked in [18] that the result is valid for general graphs, also including directed graphs. This is not true, as the example in Fig. 4 shows. There a directed path of three nodes indeed can be shattered disproving 2 as an upper bound.[2]

But as dealing with *directed* edges is naturally required when considering street graphs (one-way streets, roundabouts, asymmetric edge weights), the VC-dimension for this case is clearly of interest. We will prove in Theorem 1 that the VC-dimension for unique *directed* shortest path systems (UDSPS) is 3, and therefore we can also derive k-SPC solutions with at most $\mathcal{O}(n/k \log(n/k))$ vertices for directed graphs.

[2] Meanwhile an erratum was published by the authors of [18] which can be found here: http://www.cse.cuhk.edu.hk/~taoyf/paper/sigmod11-skip-erratum.

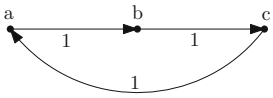

Fig. 4 Consider the shortest path node set $\pi(a, c) = \{a, b, c\}$. As $\pi(a, a) = \{a\}, \pi(b, b) = \{b\}, \pi(c, c) = \{c\}, \pi(a, b) = \{a, b\}, \pi(b, c) = \{b, c\}$ and $\pi(c, a) = \{c, a\}$, every subset of $\{a, b, c\}$ can be created by intersection with another shortest path in this graph, so the VC-dimension of the system of shortest paths (which are all unique) in this example is 3

Theorem 1 (VC-Dimension of UDSPS) *A system of unique directed shortest paths has VC-dimension at most 3.*

Proof We prove that an arbitrary set of four nodes $U' = \{v_1, v_2, v_3, v_4\}$ cannot be shattered. If there exists no shortest path containing all the nodes in U' this is trivially true. So from now on let π be a directed shortest path that contains all these nodes, w.l.o.g. in the order implied by their indices. Consider now the sets $A_1 = \{v_1, v_4\}$, $A_2 = \{v_2, v_4\}$ and $A_3 = \{v_1, v_2, v_4\}$ and assume they can be realized by paths $q_1, q_2, q_3 \in S$, meaning that $A_i = U' \cap q_i$. Because q_1 does not contain v_3 and v_2 and shortest paths are unique, it follows that v_4 is visited before v_1 in q_1, i.e., $q_1 = \cdots v_4 \cdots v_1 \cdots$. With a similar argument on q_2, we get $q_2 = \cdots v_4 \cdots v_2 \cdots$. For A_3 observe that for all possible orders of its elements in q_3—except for the order v_2, v_4, v_1—the shortest path from the first to the last node is already known and not consistent with the respective q_3, as they contain different nodes inbetween. For the remaining order, it holds that a path from v_2 to v_1 over v_4 must contain v_3, which again is not consistent with the induced q_3. Hence, there exists no q_3 and U' cannot be shattered. □

Remark 1 As already explained in [1], output-sensitive upper bounds can be derived from VC-dimension analysis, e.g., the algorithm presented in [5] yields a solution of size $\mathcal{O}(d|OPT|\log(d|OPT|))$. Plugging in our value of $d = 3$ for UDSPS, we end up with an approximation factor of $\mathcal{O}(\log|OPT|)$.

4 Constructing k-Path Covers in practice

In this section, we will develop approaches for efficient cover set construction in practice, first discussing the general case of k-Path Covers but then also considering the special case of k-Shortest-Path Covers.

A naive approach that immediately comes to mind is to enumerate for each vertex $v \in V$ all paths with k nodes that start in v and store them. Then any (heuristic) Set-Cover/HittingSet algorithm, e.g., the greedy approach, could be used to construct a feasible cover C. Unfortunately, this is not practical for large input graphs, as the exploration time as well as the space consumption of $\mathcal{O}(|V|^k)$ for extracting

and storing all k-node paths is prohibitive. Therefore, we will devise a more sophisticated approach, which allows for a considerably more efficient computation of a feasible cover C.

4.1 The Pruning algorithm

We follow a Pruning approach with the following high-level idea: Starting with all the nodes in the cover, i.e., $C = V$, we consider the nodes one by one, always deciding for a node v whether it is necessary to keep it in C to maintain the covering property. To decide whether a node v can be pruned from C, we essentially have to make sure that there exists no k-node path over v which does not contain any other node from the current cover C. So we basically have to explore all outgoing and incoming paths of v until reaching other nodes from C. If the combination of such (disjoint) outgoing and incoming paths yields a concatenated path of length k, we have to keep v in C.

A high-level view on this procedure is given in Algorithm 1.

Algorithm 1 Procedure to decide whether node v is necessary for k-APC-cover.

```
nodeNecessary(v,k)
construct the set P_o of all outgoing paths from v not containing any
    node in C − {v}
if ∃π ∈ P_o with |π| = k then
    return true
end if
for all π ∈ P_o do
    search for the longest incoming path into v
      not containing nodes in (C ∪ π) − {v}
    if such path of length k − |π| + 1 exists then
        return true
    end if
end for
return false
```

To decide whether a node v can be pruned, the procedure is called with *nodeNecessary(v,k)*. It returns true if a k-node path exists which is only covered by v (that is, v cannot be pruned) or false if no such path exists (that is, v can be pruned). The procedure uses two subroutines, enumerating all incoming and outgoing paths not containing a specific set of nodes—these subroutines can be easily implemented very similar to depth-first search (but with potentially exponential running time in k). More precisely, if $N_k(v)$ denotes the number of nodes in V which are on some k-path originating in v, then a very rough bound on the runtime can be given by $\mathcal{O}(N_k(v)^{2k})$, as there are at most

$$\binom{N_k(v)}{k-1}$$

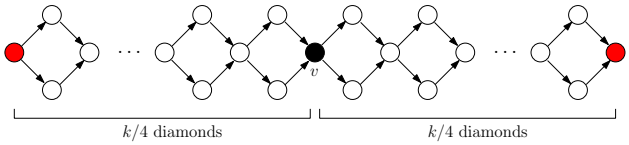

Fig. 5 To decide whether v has to be kept, $\Omega(2^{k/2})$ paths have to be considered

many outgoing/incoming paths with $\leq k$ nodes. If the maximum degree of a node is bounded by Δ then it holds $N_k(v) \leq \Delta^k$.

While the two subroutines in Algorithm 1 are naturally implemented in a recursive fashion (like depth-first search), our implementation is stack based which is much faster in practice due to the avoidance of the overhead of stack frame (de)allocation during the recursive calls.

Note that it is not difficult to come up with simple instances where this routine takes time exponential in k. Consider the problem instance in Fig. 5, where the left- and right-most (red) nodes as well as v are part of the current cover for some value k. v is $k/2$ hops away from each of the red nodes. To be sure that after the removal of v from the cover, there is no uncovered k-node path, our algorithm enumerates all uncovered incoming as well as outgoing paths (of which there are $\Omega(2^{k/4})$ each), resulting in a running time of $\Omega(2^{k/2})$. And this is not even the worst-case instance.

Still, in practice, this algorithm is pretty efficient for road networks, even though they sometimes exhibit similar patterns like the one in Fig. 5 (think of Manhattan-type structures). The reason for that is first that during the construction of the set P_o as well as the search for an incoming path the algorithm will only explore paths of length at most k since by assumption C was a valid cover before consideration of node v. More importantly, at the very beginning, when C is almost the whole vertex set, the two path enumeration steps abort almost immediately (because all uncovered paths are very short). This is the main reason for the Pruning approach to be surprisingly fast in practice.

Theorem 2 *The Pruning algorithm produces a feasible and minimal k-Path Cover C.*

Proof The Pruning algorithm only discards a node v from C if all paths of k nodes containing v are covered by $C - \{v\}$. Therefore, throughout the algorithm, we always maintain a feasible k-APC C. So especially after termination the resulting set, C has to cover all k-paths. For minimality, consider the moment when v is regarded but not pruned from C. In this case, there was a path π which contains v as the only node from the current C. Hence, any node present in the final cover C has a witness path π which certifies its necessity for the cover (see Fig. 6 for an illustration). Therefore, no node can be removed from the final C without invalidating

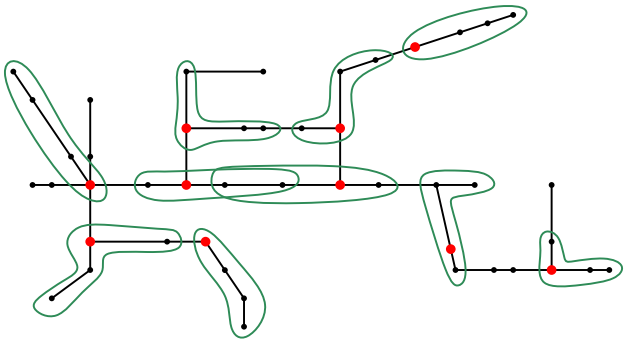

Fig. 6 A k-APC for $k = 4$ produced with the Pruning algorithm (*red nodes*). For each node in the cover, a witness path (*green*) is marked, which contains only this node from the cover. Hence, no node in the final cover can be removed without destroying the cover property

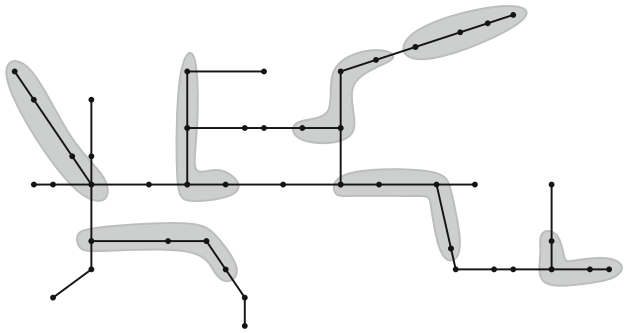

Fig. 7 Lower-bound construction for the graph in Fig. 6. The *circled* paths are disjoint and therefore prove a lower bound of seven on the size of a feasible cover for this instance. By choosing the disjoint paths more carefully, one could also prove a lower bound of ten which matches the size of the already found k-APC. Note that in general such a matching lower bound might not exist

the solution. Hence, C has to be minimal in a set theoretic sense. □

Of course, minimality of C does not imply that it is also minimum, i.e., of minimal cardinality among all possible covers. Observe that the order in which nodes are considered during the course of the algorithm highly influences the solution quality (but not feasibility or minimality). For example, by intuition, high-degree nodes cover more paths than low-degree nodes; hence, the latter should be pruned out first. We will provide an experimental study measuring the influence of different node order schemes for pruning toward the end of the paper.

4.2 Lower bounds

Unfortunately, there are no meaningful theoretic lower bounds we can compare our result to for quality analysis (as e.g., in a star graph, a single node yields a valid cover, but the solution size might be arbitrarily large). Nevertheless, for a given problem instance, we can derive valid lower bounds for practical purposes by greedily choosing disjoint k-node paths. Obviously a set of pairwise nonintersecting k-node paths requires an extra node in C for each element in this set, so the size of any such set yields a valid lower bound for the size of C. In Fig. 7 a small illustration for the lower bounding technique is provided.

To construct such a set I of nonintersecting or independent k-paths, we invoke our path enumeration algorithm as used in Algorithm 1. For every node, we store a flag indicating whether it is already part of some path in I or not. So initially all flags are set *false* as $I = \emptyset$. Then we consider the nodes in V one by one in some arbitrary order. For each node, we enumerate the k-paths containing it until we detect one that does not contain any node with a *true* flag (if such a path exists). Then this path is added to I and the flags of all contained nodes are set to *true*. As this procedure assures that no node is contained in more than one path in I, the final set I is guaranteed to contain only nonintersecting paths.

4.3 Malicious problem instances

For the considered real-world problem instances, our experiments in Sect. 7 show (via the instance-based lower bounds) that our algorithm produces results pretty close to the optimum when pruning is performed with a good ordering like by increasing node degree. So a natural question is whether one can theoretically prove (a priori and not a posteriori via the instance-based lower bounds) good approximation guarantees—our theoretic findings do not rule out the existence of a let us say 6-approximation algorithm for k-APC. Unfortunately this is not possible for our algorithm as we will see in the following. We show that our Pruning approach where nodes are considered in order of increasing degree (one of the best orderings in practice; see Table 3) can produce almost arbitrarily bad results. Consider the graph in Fig. 8, which is a central node (v_{opt}) with $d = 7$ 'arms' attached to it each of which consists of 11 nodes and the last node of each arm has additionally 7 other nodes attached to it. So the distance from v_{opt} to v_i is 11 hops each, and clearly, $C = \{v_{\text{opt}}\}$ is the optimum k-Path Cover for $k = 22$. When pruning in order of increasing degree, first all nodes except v_{opt} and $v_1, \ldots v_7$ are pruned away. But then, since v_{opt} has only degree 7, whereas the v_i all have degree 8, v_{opt} is pruned away resulting in a cover $C = \{v_1, \ldots, v_7\}$. It should be clear that one can modify this example to yield an arbitrarily large solution of our algorithm, whereas the optimum will always be of size 1. Note that the degree of these 'bad' instances becomes pretty high, so when restricting the maximum degree of the graph, some guarantee for our algorithm might be within reach. Similar bad examples can be found for the other node orderings.

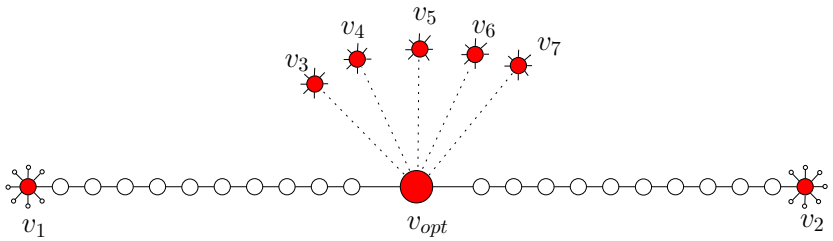

Fig. 8 Instance ($k = 22$) where our algorithm with 'increasing degree' ordering yields solution of size 7 compared to the optimum solution of size 1

4.4 Nested k-Path Covers

Reconsider our application of transmitting and visualizing hiking routes. Depending on the zoom-level in which we want to render the hiking map, we might require different values of k. So in fact we like to have a sequence of covers $C_1, \ldots, C_r$ for $k_1 < k_2 < \cdots < k_r$ to allow for r zoom-levels. Note that in this visualization context, it is crucial to demand $C_i \supseteq C_{i+1}$, because otherwise the refinement of a path when zooming-in might lead to a completely different path representation, which would make it hard for a user to recognize substructures and orient himself. To extract such a sequence of nested k-Path Covers, we first compute the cover C_1 for k_1 conventionally with our Pruning algorithm. When, for k_2 we do not initialize the Pruning algorithm with $C_2 = V$ but $C_2 = C_1$ instead. Therefore, we make sure that the resulting cover C_2 (after pruning superfluous nodes) is a subset of C_1. Correctness follows from the fact that obviously a valid cover C for some value k is always also a feasible cover for all values $k' > k$. Proceeding like this up to k_r—always taking the last computed cover as initialization for the next pruning round—we retrieve the desired sequence of nested covers.

4.5 Special case: k-Shortest-Path Cover

The problem of covering all k-node *shortest* paths for a specific metric was first tackled in [18]. The authors proposed a greedy augmentation algorithm which they call *Adaptive Sampling*. The idea is to start with an empty cover $C = \emptyset$ and then consider the nodes in V one by one, adding a node v to C iff at the moment of consideration there exists a so far uncovered k-node shortest path starting in v. Unfortunately, this approach does not guarantee minimality (in a set theoretic sense) of the resulting cover since a node v added to C at some point of the algorithm might become redundant later on due to nodes subsequently added to C.

We reimplemented their approach for evaluation, but made some small modifications which improve the quality and the runtime slightly. Even using this improved version of Adaptive Sampling, we observed that metric-independent k-Path Covers constructed with our Pruning algorithm were smaller in size than the k-SPC by Adaptive Sampling on the same graph (even though in the k-SPC setting much fewer paths have to be hit).

4.5.1 Quick Pruning

To improve further, we adapted the Pruning algorithm to the k-SPC setting. Like for the general case we start with $C = V$ and try to prune nodes ensuring that their removal does not lead to any uncovered k-node shortest path.

To decide if there exists an uncovered k-node shortest path containing v, we proceed as follows:

1. temporarily remove v from C
2. grow a shortest path tree $T_F(v)$ by running Dijkstra's algorithm until all unsettled but labeled nodes contain a node from C on their current path from v
3. in the reversed graph[3] G^{-1} grow a shortest path tree $T_R(v)$ from v until all unsettled but labeled nodes contain a node from C on their current path from v
4. if $T_F(v)$ contains a k_F-node path not containing any node from C and $T_R(v)$ a k_R-node path not containing any node from C and $k_F + k_R - 1 \geq k$, add v back to C otherwise prune it.

We call this algorithm *Quick Pruning* because it runs very fast in practice. The theoretic runtime can be bounded by $\mathcal{O}(N_k(v)^2)$ where $N_k(v)$ denotes again the number of nodes in V that lie on a k-path originating in v. As there are at most $N_k(v)^2$ edges between those nodes, a single Dijkstra runs costs $\mathcal{O}(N_k(v) \log N_k(v) + N_k(v)^2) \in \mathcal{O}(N_k(v)^2)$ which determines the runtime.

But in contrast to our general Pruning algorithm for k-APC, we cannot guarantee minimality with this approach (the same holds for *Adaptive Sampling*). The reason for possibly keeping some unnecessary nodes in C is that the concatenation of two shortest paths (one from T_R, one from T_F) not necessarily needs to be a shortest path itself. So the k-node path we take as a witness for the necessity of v might not be a shortest path and therefore does not have to be covered by our C. In fact, we can fix this by running a slightly modified Pruning algorithm.

[3] G^{-1} has the same vertex set as G but all edges reversed.

4.5.2 Pruning for k-SPC

For every node in the backward search tree T_R, we run a forward search and check if there are uncovered k-node shortest paths over v. More formally it can be described like this:

1. temporarily remove v from C
2. in the reversed graph G^{-1} grow a shortest path tree $T_R(v)$ from v until all unsettled but labeled nodes contain a node from C on their current path from v
3. for every node w in $T_R(v)$ grow a shortest path tree $T_F(w)$ in G until all unsettled but labeled nodes contain a node from C on their current path from w
4. if for some w there is a k-node path over v in $T_F(w)$ not containing a node from C, v has to be added back into C, otherwise it can be pruned

While this Pruning approach again guarantees set minimality of the output cover, it triggers a lot more Dijkstra computations. Theoretically, the runtime increases from $\mathcal{O}(N_k(v)^2)$ for Quick Pruning to $\mathcal{O}(N_k(v)^4)$.

4.5.3 Upper bounding technique

The following observation can save some of the tree constructions in the Pruning algorithm: Let $u_1, u_2, \ldots u_j$ be the neighbors of a node w on shortest paths from w to v (if the shortest path is unique then $j = 1$). For $x \in V$, let $U(x)$ be an upper bound on the number of nodes of a shortest path free from nodes of C starting in x. If $U(u_i) \leq l$ for all u_i, $U(w) \leq l+1$. Hence, we can examine nodes in T_R in increasing hop distance from v. Initially $U(x) = \infty, \forall x \in T_R$. For a node w to be examined $U(w) := max_{u_i} U(u_i) + 1$ for all its neighbors u_i on shortest paths to v. If $U(w) < k$ the construction of $T_F(w)$ can be skipped, otherwise we compute $T_F(w)$ and set $U(w)$ accordingly. This exploration strategy saves many Dijkstra computations as we will see in the experimental section.

Still, even when applying the upper bounding technique, we expect the runtime to be worse than for Quick Pruning—but the quality to be superior. So it depends on the application context which approach to use.

4.5.4 Dealing with ambiguous shortest paths

According to our definition of k-SPC, we aim at covering *all* shortest paths, not only one shortest path for each pair s, t of vertices. We now provide the details which we left out in the above description for the sake of a clearer presentation. The basic idea is to temporarily make shortest paths already covered by the current C infinitesimally more expensive such that uncovered shortest paths are always exhibited. To that end, consider slightly modified edge costs $c' : E \to \mathbb{R}_0^+$ where for an edge $e = (v, w)$ with $v \in C$ or $w \in C$ we define $c'(e) = c(e) + \epsilon$ for some arbitrarily small $\epsilon > 0$, otherwise $c'(e) = c(e)$. Growing a shortest path tree from s under this edge cost function c' until all nodes have a node from C on their shortest path from s ensures that if there exists a shortest path from s to v not containing any node from C, its nodes will be part of the shortest path tree grown from s. Now consider the directed acyclic graph D induced by the nodes of the shortest path tree and all edges $e = (v, w)$ with $d(v) + c(e) = d(w)$ [here $d(.)$ denotes the shortest path distance from s with respect to c]. Every path in D from s to some node v corresponds to a shortest path from s to v not containing any nodes from C and vice versa. The maximum-hop path among these can easily be determined for all nodes in D in $O(|D|)$ time.

We want to emphasize, though, that this is only necessary if we really insist on hitting *all* shortest paths. It is very easy to enforce uniqueness of shortest paths by techniques like symbolic perturbation. In practice, the ambiguity of shortest paths hardly affects the size of the covers in road networks.

5 Overlay graph construction

For some applications (as map simplification or routing), we not only are interested in the set of nodes C which form the k-SPC or k-APC but care for constructing the induced overlay graph $G_O(C, E_O)$ as well. So for every $v \in C$, we have to insert edges to all of its k-Cover neighbors. Here, a node $w \in C$ is a neighbor of v, if there exists a (shortest) path between them which does not contain further nodes from C.

In [18] an overlay graph construction algorithm was described for dealing with a k-SPC. We will first briefly review this approach and point out several possible improvements. Then we describe a simple and efficient algorithm to construct the induced overlay graph for a k-APC.

5.1 k-SPC Overlays

To extract the set of overlay edges E_O, a two-step procedure was proposed in [18]: First, for every node $v \in C$ the k-hop shortest path tree $T_k(v)$ is computed. Then all paths rooted in v in $T_k(v)$ get traversed up to the point where a node $w \in C$ is passed, and then the edge $\{v, w\}$ is added to E_O. This approach has some drawbacks:

– As already stated in the original paper, superfluous edges are likely to be added, increasing the final graph size and query times later on. The reason is that there might exist shorter paths with a larger hop distance to nodes in $T_k(v)$; hence, the contained nodes might not actually be in the neighborhood of v (see Fig. 9 for an example).

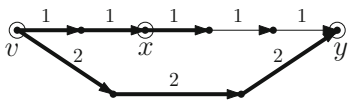

Fig. 9 Neighborhood extraction for $k = 3$: The 3-hop shortest path tree $T_3(v)$ is printed in *bold*, *circled* nodes are part of the k-SPC. Here, the edges (v, x) and (v, y) would be added, while only (v, x) is necessary

– The method requires always the computation of the complete k-hop shortest path tree for every node, but the neighbors might be considerably closer.

In the following, we present approaches that compute the *exact* set of edges E_O for a given k-SPC efficiently in practice.

5.1.1 Neighborhood extracting Dijkstra

To use Dijkstra efficiently for computing the set of neighbors of a node v, we need to stop its execution as soon as all neighbors of v are settled. That point is certainly reached when all nodes still in the priority queue contain at least one node from C on their tentative path from v. So our Dijkstra's search radius never exceeds the distance of the furthest neighbor. In particular in the case where all neighbors are close by, this improves the runtime compared to the approach presented in [18] significantly. Also no superfluous edges are inserted.

5.1.2 CH-based neighborhoods

The construction of an overlay graph for given k-SPC C could also be interpreted as removing all nodes V\C from the graph while preserving the distances between the remaining ones. This captures essentially the idea of the shortest path speedup technique *contraction hierarchies* [12] (CH). Here, in a preprocessing phase nodes are removed from the graph (contracted) one by one. To maintain the shortest path distances between all so far not contracted nodes, so called shortcut edges (a.k.a. shortcuts) are added whenever necessary. A shortcut edge (u, w) is required if at the moment of contraction of a node v the path u, v, w is a shortest path. So a Dijkstra computation reveals which shortcuts to add for every pair of adjacent nodes of v in the actual graph. After all nodes are contracted, the original graph G is augmented with all shortcut edges created during the contraction phase. In the resulting graph, many long shortest path sections are spanned by single shortcuts. A variant of Dijkstra's algorithm is able to take advantage of these shortcuts, yielding a significantly smaller search space than the conventional Dijkstra (without CH) for the same s-t-query.

In many applications, the contraction process is not fully completed (as maybe this would result in too many shortcuts in the final graph); see, e.g., [8]. The set of uncontracted nodes is then called the core of the graph. At the moment the contraction process stops, the actual graph consists of the core nodes and all edges between them necessary to guarantee the same pairwise shortest path distances as in the original graph. This is exactly what we want to achieve when constructing the overlay graph. Hence, declaring C the set of core nodes and applying the CH construction to our graph results in the correct induced overlay. The advantage of this method is that it only requires the use of the conventional Dijkstra algorithm without any adaptions.

If we use a nested construction for C, CH-based overlay graph construction exhibits another advantage. In particular, we can compute the overlay graph for all intermediate covers $C_1, C_2, \ldots, C_r = C$ in a single contraction phase. For this purpose, we first contract all nodes that are in none of the covers. This results in an overlay graph for C_1. Next we contract all nodes in the actual graph that are exclusively contained in C_1. As all remaining nodes in the graph must form C_2, we now have an overlay graph for this cover. We proceed always contracting nodes that only appear in the next cover, until we finished creating the overlay graph for C. Besides being economical to construct all these overlay graphs at once, also the zoom-in operation (i.e., refinement of paths) can be made more efficient with this approach. During the CH construction, we store for every shortcut edge the IDs of the two (shortcut or original) edges it directly bridges (in the explanation above, the edge (u, w) would point to (u, v) and (v, w)). So if we want to refine a shortcut, we can just 'unpack' it by replacing the shortcut with the two edges it spans. This can be continued recursively, allowing to zoom in to any desired level of detail without the need to run an additional local Dijkstra computation as in the approach described in [18].

5.2 k-APC Overlays

Constructing the overlay graph for a k-APC is somewhat easier as for k-SPC, as we do not have to take care of the shortest path characteristic. We present a simple breadth first search (BFS)-based approach, and subsequently also a way for constructing nested overlay graphs efficiently.

5.2.1 Neighborhood extracting BFS

To construct G_O for a given k-APC, we proceed as follows: We run breadth first search on G from each node $v \in C$. Every time a node w which is in $C - \{v\}$ is extracted from the queue, we add the respective edge (v, w) to E_O, but do not relax outgoing edges of w. So we never explore paths that are already hit by C. Therefore, our algorithm terminates as soon as all paths end with nodes in C (which due to the characteristic of C being a k-APC happens after at most all

nodes which are $k-1$ nodes away were visited; but possibly much earlier).

5.2.2 Customizable CH-based neighborhoods

Very recently, a new variant of CH was introduced [9] which inserts shortcuts not on the basis of some predefined edge metric, but purely based on the topology of the graph. So when contracting a node v, shortcuts between all pairs of direct neighbors of v are inserted. The advantage of this approach is that the constructed CH-graph can be coated with an arbitrary metric afterward and optimal query answering can still be guaranteed. This allows to switch the metric on demand without having to recompute the CH-graph from scratch. As the metric can be chosen individually, this kind of CH is also called a customizable CH. We can take advantage of this CH construction scheme when dealing with nested covers in a very similar way than described for k-SPC. Again, we always iteratively contract away all nodes besides the ones in the cover C_i. As all shortcuts between the remaining nodes in the core were inserted during the contraction process, we have a valid overlay graph for C_i at hand.

6 Application: personalized route planning

As mentioned in the introduction, we consider k-APC a basis for speeding up fully personalized route planning queries. As input we are given a road network $G(V, E)$ and a set of cost functions $c_1, c_2, \ldots c_r$ with $c_i : E \to \mathbb{R}_0^+$. A query consists of source node s, destination node t and weights $w_1, w_2, \ldots, w_r$ with $w_i \in \mathbb{R}_0^+$ and expects as a result a path π minimizing the weighted cost $\sum_{e\in\pi} \sum_{i=1}^{r} w_i \cdot c_i(e)$. The straightforward baseline strategy to answer such a query is to run Dijkstra's algorithm and each time an edge is considered in the course of the algorithm, compute its respective weighted cost according to the w_i values provided with the query.

Apart from [11], which is only practicable for a small number of metrics ($r \leq 3$) because the preprocessing becomes too complicated and time-intensive otherwise, we are not aware of any speedup scheme for such type of queries. The customizable route planning approach in [6] allows for updates of the underlying graph metric, but an update takes several CPU core seconds on fast machine and hence is only worth if several queries with exactly the same weights are to be answered. Our approach allows for the specification of different weights w_i with every single query. In the following, we describe the details of the preprocessing phase as well as the query answering algorithm.

In the preprocessing phase, we compute auxiliary information to augment the original graph such that subsequent fully personalized route planning queries can be answered more quickly. This phase consists of the following three steps:

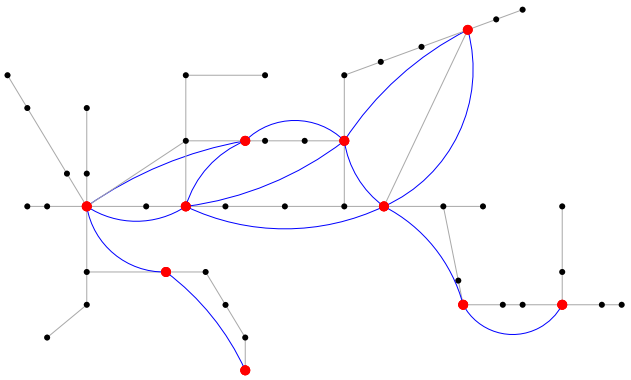

Fig. 10 4-Path Cover (*red*) and induced overlay graph (*blue*)

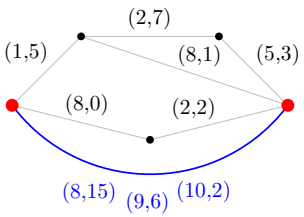

Fig. 11 Example graph with two metrics ($r = 2$). Cover nodes are colored *red*, the overlay graph consisting of a single edge is *blue*. As there are three different simple paths from one cover node to the other, the overlay edge between them gets assigned three labels. The overlay labels stem from summing up the original labels along the respective paths

- Compute a k-APC C.
- Construct the *overlay graph* $G_O(C, E_O)$ with respect to $G(V, E)$ and C as described in the last section. An illustration can be found in Fig. 10.
- Assign to each edge a set of cost vectors. In fact, an edge $(v, w) \in E_O$ gets assigned as many cost vectors as there are simple paths between v and w in G which do not contain further nodes from C. The r values of a cost vector arise from component-wise addition of the cost values of the edges of the respective path. See Fig. 11 for an illustration.

In the query answering phase, every user specifies s, t and $w = w_1, \ldots, w_r$ according to his preferences. As source and target do not have to be nodes in the k-APC C, we have to take care of connecting them to the overlay graph at first. This results in the following routine for query answering:

- Start a Dijkstra (using edge cost functions $c_1, \ldots, c_r$ and weights $w_1, \ldots, w_r$ according to the query) in $G(V, E)$ from $s \in V$ stopping as soon as all nodes in the priority queue have the shortest paths from s containing at least

one node from C. If this search has already settled t, we are done. Otherwise, we obtain a set of *access nodes* $A(s) \subset C$ as well as their shortest path distances $d(s, a_s)$ for all $a_s \in A(s)$. We know that the shortest path from s to t has to pass through one of the nodes in $A(s)$.
- Do the same from the target t, but on the reverse graph G^{-1}; this yields a set of access nodes $A(t) \subset C$ as well as shortest path distances $d(a_t, t)$ for all $a_t \in A(t)$. The shortest path from s to t has to pass through one of the nodes from $A(t)$.
- In the overlay graph $G_O(C, E_O)$ fill the priority queue with the nodes $a_s \in A(s)$ (initialized with the respective distances $d(s, a_s)$ and let Dijkstra run until all nodes in $A(t)$ have been settled. During edge relaxation, all cost vectors assigned to the edge have to be considered, and the weights w_i specified in the query determine which cost vector leads to the smallest edge cost. After the Dijkstra run finished, we have found shortest path distances $d(s, a_t)$ from s to all nodes $a_t \in A(t)$. The desired shortest path distance (and the path itself) is determined by the access node $a_t \in A(t)$ minimizing $d(s, a_t) + d(a_t, t)$.

7 Experimental evaluation

7.1 Environment and data sets

Our C++ implementations were compiled using gcc 4.6.3 and benchmarked on a 3.2 GHz intel i5-3470 with 16 GB RAM. As primary benchmark data, we used road networks extracted from the OpenStreetMap (OSM) project as well as the DIMACS road network graphs[4] that are freely available and which were also used in [18]. Edge costs (in the single metric case) were set to travel times. Most experiments were conducted on the largest graphs GER and USA; see Table 1 for an overview of the characteristics of the used graphs. The values of d_{avg}, $d_{\max}$ and t_{dij} denote the average degree, the maximum degree and the time a random one to all run of Dijkstra's algorithm takes on average, respectively.

For a comparison with the experimental results on the MGS problem reported in [17], we extracted the same benchmarks used there from the Stanford Large Network Dataset Collection[5] and some additional (bigger) networks. The network types include social networks (Wiki-Vote, LiveJournal1), collaboration networks (CA-GrQc, CA-HepTh), peer-to-peer networks (P2P**), product networks (Amazon0601), web networks (web-Google) and community networks (com-lj). An overview of the characteristics of the used data sets is provided in Table 2.

[4] http://www.dis.uniroma1.it/challenge9/download.shtml.

[5] http://snap.stanford.edu/data/.

Table 1 Features of the benchmark graphs based on real-world road networks ($M = 10^6$)

Name	#Nodes	#Edges	d_{avg}	$d_{\max}$	t_{dij} (ms)
DIM					
CAL	1.89M	4.65M	2.46	8	139
USA	23.95M	58.33M	2.43	9	3142
OSM					
BW	2.23M	4.64M	2.04	7	396
GER	17.73M	36.06M	2.03	8	3823

Table 2 Features of social and related networks used for benchmarking. The upper part of the table contains the networks also used in [17]. 'D' indicates the graph diameter

Name	#Nodes	#Edges	d_{avg}	$d_{\max}$	D
CA-GrQc	5242	28,980	11.06	81	17
CA-HepTh	9877	51,971	10.52	65	17
Wiki-Vote	7115	103,689	29.15	893	7
P2PG08	6301	20,777	6.59	48	9
P2PG09	8114	26,013	6.41	61	10
P2PG30	36,682	88,328	4.82	54	10
P2PG31	62,586	147,892	4.72	78	11
Amazon0601	0.40M	3.39M	16.79	10	21
web-Google	0.88M	5.11M	11.65	456	21
com-lj	3.99M	34.68M	17.34	14,703	17
LiveJournal1	4.84M	68.99M	28.46	20,293	16

7.2 Constructing k-APCs

Let us start with the Pruning approach for constructing sets C covering *all* paths consisting of k nodes: In Table 3, we first examine how different node orders affect the quality and the running time of the cover construction. We considered the two largest networks GER and USA; fixing $k = 16$, we evaluated node orders both decreasing (`-dec`) and increasing (`-inc`) according to their node ID (`id-`) as given by the original graph file, number of incoming plus outgoing edges (`oi-`), the order in which the recursive calls of a depth-first-search visit the nodes (`dfs-`), the order in which the recursive calls of a depth-first-search are completed on the nodes (`comp-`) and finally simply random order (`rand`). Intuitively it makes sense to prune away low-degree nodes first, and indeed pruning in increasing degree order (`oi-inc`) leads to much smaller cover sizes compared to `oi-dec`. The orders `dfs-dec` and `comp-inc` tend to prune out nodes in dead-ends first which seems favorable to `dfs-inc` and `comp-dec`. For the lower bounds, the differences are not very pronounced, so throughout the following benchmarks, we use the `comp-inc` order for both cover construction and lower bounds. Note that our instance-based

Table 3 k-APC: Influence of different node orders on cover size and lower bounds for $k = 16$

G	Order	Lower bound		Cover		
		Size	Time (s)	$\|C\|$	$\|C\|/\|V\|$ (%)	Time (s)
GER	id-inc	681,242	10	1,302,559	7.35	35
	id-dec	628,533	12	1,992,315	11.20	48
	oi-inc	708,417	11	1,368,034	7.72	39
	oi-dec	622,685	13	2,072,841	11.70	56
	rand	659,958	21	1,740,967	9.82	73
	dfs-inc	735,199	10	1,913,269	10.80	64
	dfs-dec	727,341	10	**1,201,654**	6.78	47
	comp-inc	735,746	10	1,209,215	6.82	46
	comp-dec	736,775	10	1,877,547	10.60	62
USA	id-inc	720,221	9	3,232,581	13.50	98
	id-dec	739,922	10	2,951,161	12.30	79
	oi-inc	764,760	12	2,504,626	10.50	61
	oi-dec	685,110	13	4,191,685	17.50	224
	rand	716,058	28	3,112,202	13.00	134
	dfs-inc	755,352	11	3,674,978	15.30	137
	dfs-dec	752,906	11	**2,351,124**	9.82	110
	comp-inc	759,961	11	**2,351,124**	9.82	111
	comp-dec	755,338	11	3,704,711	15.50	151

Bold values indicate the (smallest) cover sizes for GER and USA respectively (among all different node orderings that were tested)

lower bounds prove that our constructed covers are pretty close to optimal (at most a factor 1.7 larger for GER, a bit worse with a factor of 3.2 for USA)—in fact, they could be even closer to the optimum since the lower bound is probably not really tight. The construction times for the lower bounds are almost negligible.

In Table 4, we examine the cover construction for varying values of k. As to be expected, for growing k the cover construction time increases rapidly; nevertheless, it is somewhat astonishing that it is feasible to construct covers for k values as large as $k = 32$. Also note that while the lower bounds of the GER and USA graphs are very similar, the cover sizes (and the respective construction times) are considerably worse for the USA graph—one reason might be the presence of many grid-like substructures in the USA road network. Nevertheless, the approximation ratio guaranteed by our instance-based lower bounds never exceeded 2.2 (for GER) and 5.0 (for USA); the actual optimum might also be much closer to our constructed covers than to our rather naive lower bound.

7.3 Special case: constructing k-SPCs

For comparison with the results in [18], we implemented a variant of their Adaptive Sampling approach and our two pruning strategies for the k-SPC case. The respective results can be found in Table 5.

The outcomes of our implementation of the Adaptive Sampling approach are pretty close to their reported performances (even slightly better in terms of quality of the solution): for the USA instance and $k = 16$ our implementation of Adaptive Sampling constructed a cover of size 3,295,812 which is about 14 % of the total number of nodes in the graph ([18] reported around 15 % for this very instance). Our Pruning approach on the other hand produced for the same instance and k-value a cover of size only 5.8 % of the total number of nodes. For all choices of k the Pruning approach consistently outperformed Adaptive Sampling by a considerable margin in terms of quality.

Concerning running time, we described an upper bounding technique to save Dijkstra computations. In Fig. 12, we collected data implying how effective this technique is; exemplary for the `BW` road network and node order `comp-inc`. While for small values of k, the gain of the upper bounding technique is almost nonexistent, for larger k reduction in Dijkstra computations is directly reflected in the running times. Already for $k = 40$ the upper bounding technique leads to more than 4 times faster running times; this gap is expected to increase with growing k. Therefore, in all our experiments, we employed this upper bounding technique.

Running times of Pruning are slightly above the ones for Adaptive Sampling, though, yet our Quick Pruning variant (which does not guarantee minimality) was always much

Table 4 k-APC: Approximation ratios (apx) and construction times for comp-inc order and varying values of k. 'Percent' describes the fraction of nodes in V that are contained in the cover C

G	k	Lower bound	$\|C\|$	Percent (%)	Time (s)	apx
GER	2	8,560,543	8,863,443	50.00	17	1.04
	4	3,969,092	4,513,217	25.50	21	1.14
	8	1,739,476	2,308,934	13.00	29	1.33
	16	735,746	1,209,215	6.82	47	1.64
	32	306,009	666,829	3.76	119	2.18
USA	2	10,906,996	11,910,322	49.70	15	1.09
	4	4,631,511	6,676,239	27.90	22	1.44
	8	1,854,605	3,776,360	15.80	38	2.04
	16	759,961	2,351,124	9.82	110	3.09
	32	321,853	1,603,267	6.69	15,100	4.98

Table 5 k-SPC: Comparison of Sampling and Pruning and the Quick Pruning variant on the USA and GER graph for varying values of k

G	k	Lower bound	Adaptive Sampling		Quick Pruning		Pruning	
			$\|C\|$	Time (s)	$\|C\|$	Time (s)	$\|C\|$	Time (s)
USA	8	1,750,150	5,483,792	172	4,563,885	76	3,067,632	175
	16	618,755	3,295,812	615	2,449,744	179	1,392,803	782
	32	200,774	1,890,620	2700	1,256,871	513	584,904	4970
	40	136,592	1,564,624	4401	1,004,268	772	431,686	9520
GER	8	1,637,613	3,659,568	92	3,294,225	42	2,184,986	76
	16	654,679	2,142,327	248	1,710,510	89	1,028,696	222
	32	246,459	1,259,841	783	826,636	210	463,064	856
	40	177,619	1,065,534	1179	648,508	322	355,062	1360
	48	135,545	925,359	1584	530,949	433	285,780	2050
	56	107,687	825,694	2214	447,816	546	237,479	2900

faster than Adaptive Sampling but still better in terms of quality of the solution. So it outperforms Adaptive Sampling in all aspects.

In general—for a given time budget—k-SPC can be computed for larger values of k, in comparison with k-APC, which does not come as a real surprise since at some point considering all possible paths of some length starts exhibiting exponential blowup.

It is worth emphasizing that e.g., for $k = 16$ our pruning-based k-APC covers which guarantee covering *all* k-paths have smaller cardinality than the Adaptive Sampling-based k-SPC covers (like [18]) which only cover *shortest* k-paths in spite of the much stronger coverage property (in the USA instance: 2,351,124 vs. 3,295,812 nodes).

In Fig. 13 we compare the k-APC and k-SPC results for GER and growing values of k. Not too surprisingly, for very small k, a k-APC is not much larger than a k-SPC, since there are simply no that many potential nonshortest paths. For growing k, though, the number of nonshortest paths naturally grows faster than the number of shortest paths; hence, the gap between an k-APC and k-SPC increases.

In addition, we analyzed the hop distances between neighboring nodes for AS and Pruning, exemplary for USA with $k = 16$. The average hop distance between two cover nodes for AS is 6.45, for Pruning 8.04. Even more significant is the distribution of hop distances among all neighbors as depicted in Fig. 14. For AS we observe that a large number of neighboring nodes are only a few hops apart. Indeed, the maximum is at 1, and the curve decreases almost monotonously with growing hop distance. This is a direct consequence of the redundancy of nodes in an AS cover. However, for Pruning, the curve looks quite different: The peak is at 8 and—as there is also the median—a significant number of neighbor pairs has a larger hop distance.

We get a very similar result when studying k-Shortest-Path queries. Here, given a source and a target vertex $s, t \in V$, we want to compute the subsampled representation of a shortest path from s to t. These numbers indicate the compression of shortest paths using the k-SPC approach. In Fig. 15 the distribution of hop distances between two consecutive nodes on such k-paths is depicted (again for the USA graph and $k = 16$) using a set of randomly chosen s, t pairs. In comparison with Fig. 14, we notice for both distributions (AS and Pruning) a small rise for $k = 16$ in the path evaluation. This could result from many important streets (like highways)—which are very likely to be part of a shortest path for random source and target—consisting of long chains of nodes with degree 2. For such chains, the local optimal cover consists of

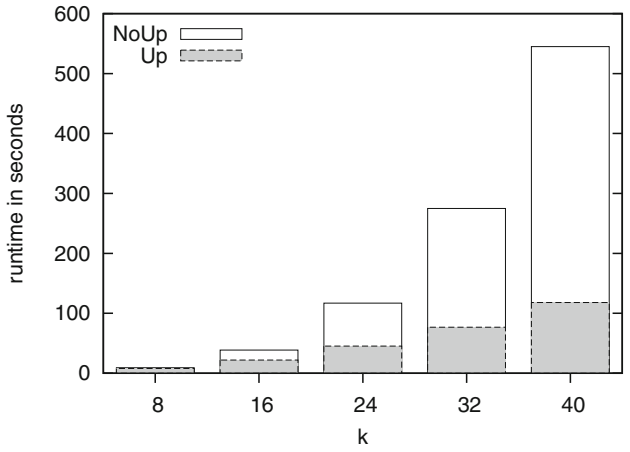

Fig. 12 k-SPC-cover: Effect of upper bounding technique on the running time (order: `comp-inc`)

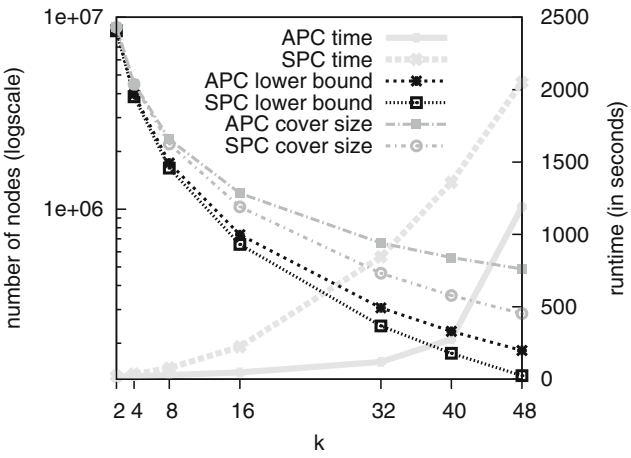

Fig. 13 k-SPC- versus k-APC-cover on the GER graph

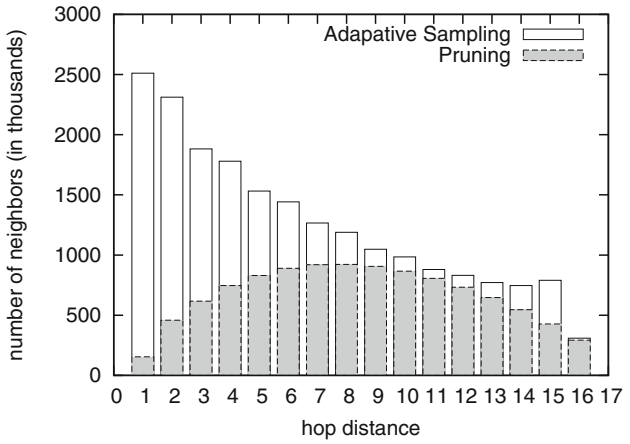

Fig. 14 Distribution of hop distances between neighbored nodes in the USA overlay graph for $k = 16$

nodes being k apart. If the actual cover comes close to this sampling, the relatively high frequency of path sections with a hop distance of 16 in the plot seems natural.

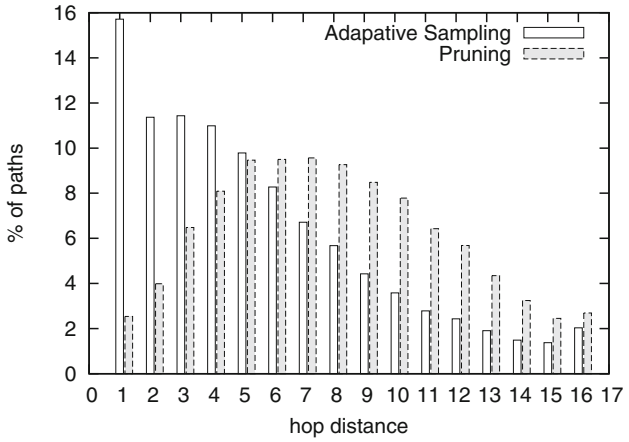

Fig. 15 Distribution of hop distances between nodes of C for 1000 random shortest paths on the USA overlay graph for $k = 16$

Table 6 Ratio of edges on a subsampled path compared to all edges in the respective original path in percent. Values are averaged over 1000 random queries

k	2 (%)	4 (%)	8 (%)	16 (%)	32 (%)
AS	67	42	26	18	12
Pruning	56	33	20	12	8
Lower Bound	50	25	12	6	3

The resulting number of edges in the subsampled representation of a path compared to the number of edges in the original graph can be found in Table 6 for varying values of k. A natural lower bound for the number of edges on such a path is the number of original edges divided by k (otherwise, the underlying cover would not be a valid k-SPC). If we compare AS and Pruning, we see that Pruning results in edge values much closer to the lower bound than AS, and especially for small k Pruning is near-optimal.

7.4 Nested/hierarchical construction

Constructing C for given k hierarchically, i.e., picking a subset of a cover for a smaller parameter k', promises a runtime improvement due to smaller search radii in the initial exploration phase and fewer nodes to consider later on. On the other hand, it is not clear that a subset of a prior cover can be a close-to-optimal cover for k. These observations are also reflected in our experiments. We constructed the k-SPC and the k-APC hierarchically for the USA graph and compared the runtime and the quality to the direct construction; see Table 7.

We observe that the runtime decreases significantly with the hierarchical approach (especially for larger values of k). We pay for this efficiency with slightly increased cover set sizes. Still a maximum of 8.1 % more nodes in the k-SPC

Table 7 Hierarchical construction of cover sets for the USA graph. Percentages denote increase/decrease compared to the computation time and cover set size when using the Pruning algorithm for each value of k individually

k	2 (%)	4 (%)	8 (%)	16 (%)	32 (%)
k-SPC					
Δ time	+0	−6.94	−63.21	−60.31	−83.97
Δ size	+0	+0.04	+8.10	+7.2	+1.58
k-APC					
Δ time	+0	−17.41	−49.50	−63.89	−84.99
Δ size	+0	+3.30	+8.40	+8.12	+6.00

(8.4 % in the k-APC) for $k = 8$ compared to the result of the conventional Pruning approach is tolerable; and as with increasing k this deviation shrinks, we consider hierarchical cover retrieval a useful speedup technique.

7.5 Overlay graph computation

We described three different methods to compute the overlay graph upon a k-SPC: the tree-based (TB) approach from [18], the Dijkstra-based (DB) approach and the CH-based (CHB) approach. We compared these three approaches in terms of run time and edge ratio (i.e., number of created edges divided by the number of original edges in G, abbreviated ER) on the example of the USA graph, with the input cover derived from the Pruning method. The respective results are summarized in Table 8.

We observe that for small k the TB algorithm is the fastest, while for larger k it is clearly outperformed by our Dijkstra-based approach. For DB we see only a linear grow of the runtime with increasing k; hence, DB seems applicable for even larger values of k than the ones considered here. For CHB, the runtimes are not really comparable to the other approaches, as here the input sets were constructed hierarchically and are therefore larger. Moreover, CHB constructs the overlay graph not only for the respective k-value in the row, but also for all smaller ones listed in the table. For small values of k, the summed runtime of computing all these overlay graphs individually with DB is still below the runtime for CHB (note that the runtimes for DB in Table 8 are on other inputs, not the hierarchical ones necessary for CHB). Only for $k \geq 32$, we observed an improved runtime with CHB. But remember that CHB overlay graph construction allows for more efficient zoom-in operations in a straightforward way.

Table 8 Overlay graph construction with three different approaches (*TB* tree based, *DB* Dijkstra based, *CHB* Contraction Hierarchy based) for the USA graph. ER denotes the edge ratio, timings are given in seconds

k	TB		DB		CHB	
	Time	ER	Time	ER	Time	ER
2	18	66.08	22	64.74	52	64.74
4	22	45.33	37	44.14	100	46.57
8	33	29.44	46	28.51	163	31.16
16	185	18.45	82	17.71	253	18.61
32	6742	10.21	143	9.95	366	9.56

Table 9 Edge ratios for k-SPC-based overlay graphs created with DB along with the time for $k = 32$ (in seconds), which upper bounds the time for all smaller values of k

k	2	4	8	16	32	Time
BW	54.77	31.58	18.29	10.35	5.57	8
GER	55.12	31.91	18.37	10.39	5.69	86
CAL	66.94	46.84	30.60	19.06	10.50	13
USA	64.74	44.14	28.51	17.70	9.95	143

In terms of quality, the TB and the DB approach differ only slightly and the effect decreases with growing k. This is mostly due to our cover sets C are so tight that the addition of superfluous edges in the TB approach is inhibited to a great extent.

We also collected the edge ratios for all our test graphs resulting from the DB approach in Table 9. In coherence with the smaller covers, the OSM graphs also exhibit much smaller edge ratios than the DIMACS graphs.

Note that it is crucial on which sets such an overlay graph is constructed. As described above, the Pruning method results in much smaller cover sets than Adaptive Sampling. We observed that these much smaller covers also yield significantly smaller edge sets in the overlay graph (independently of the construction method for the overlay). In [18], the edge ratio for $k = 4$ was 61 % (while ours is about 50 %), for $k = 8$ they reported 47 % (compared to >38 % by us) and for $k = 16$ they got 38 % (only about 20 % in our evaluation). So the total compression of the USA graph choosing $k = 16$ is 0.17 with our methods (i.e., 17 % of nodes and edges combined remain), while only 0.31 in [18].

7.6 Personalized route planning

Our scheme for answering personalized route planning queries uses the overlay graph induced by a concise k-APC. The efficiency of our approach is based on the fact that the search in the overlay graph $G_O(C, E_O)$ turns out to be much faster than on the original graph $G(V, E)$ and the initial (local) searches for $A(s)$ and $A(t)$ take negligible time.

In the following, we present detailed experiments regarding the preprocessing and the query answering phase.

7.6.1 Personalized route planning: preprocessing

Let us first experimentally examine the cost and structure of the overlay graph $G_O(C, E_O)$.

Table 10 Personalized route planning/preprocessing. Structure and construction of the overlay graph

k	$\|C\|$	Edges			Cost vectors			Time (s)
		Total	Avg. degree	Max degree	Total	Average/edge	Max/edge	overlay
GER								
4	4,513,217	11,968,468	2.65	16	12,525,519	1.05	9	42.1
8	2,308,934	7,628,636	3.30	25	9,117,057	1.20	35	36.8
12	1,573,537	6,101,729	3.88	28	8,740,617	1.43	106	39.4
16	1,209,215	5,321,096	4.40	32	9,504,005	1.79	408	45.2
20	990,560	4,848,058	4.89	43	11,441,398	2.36	6477	55.6
24	845,905	4,532,018	5.36	40	15,050,313	3.32	7652	76.0
28	743,694	4,308,376	5.79	38	22,100,074	5.13	48,470	119.2
32	666,829	4,143,342	6.21	40	32,057,249	7.74	67,537	183.3
36	607,151	4,014,888	6.61	46	64,058,939	15.96	608,762	395.6
40	559,502	3,911,824	6.99	47	137,496,018	35.15	1,746,242	950.2
44	520,237	3,829,285	7.36	54	396,197,570	103.47	7,741,250	3208.6

In Table 10, we see the characteristics of the overlay graphs constructed for our largest OSM benchmark graph of Germany and different values of k. The overlay graph construction itself is independent of the chosen metrics. The number of cost vectors per overlay edge (v, w) with $v, w \in C$ coincides with the number of simple paths between v and w which do not contain further nodes from C. The vectors themselves are then the result of component-wise summation of the cost vectors assigned to the original edges. Since this preprocessing only has to be performed if the combinatorial structure of the road network changes (addition/deletion of nodes or edges), the respective running time is not that important. Nevertheless, we see that even large graphs like GER can easily be preprocessed within a few minutes for moderate values of k. For too large values of k, the number of cost vectors explodes, though.

7.6.2 Personalized route planning: query processing

To realize personalized route planning queries, we have to coat the overlay graph with actual cost vectors. We considered the 8 metrics described in Table 11.

For query evaluation, we generated 100 random queries, i.e., random source and target vertex and random weights $w_1, w_2, \ldots, w_8$, and compared the performance of our speedup scheme for different values of k with a standard Dijkstra run. Apart from the measured query times, we also look more closely at the time spent in the search for the access nodes $A(s)$ and $A(t)$ as well as the search in the overlay graph. See Table 12 for the results.

We first observe that the searches for the access node sets $A(s)$ and $A(t)$ in the original graph only take negligible time, and the search in the overlay graph clearly dominates the overall query time. The achieved speedup first quickly grows with increasing k, but improves only slightly above $k = 20$. One should bear in mind, though, that the overlay graph gets very dense for large k (see Table 10), so in terms of memory efficiency, it is not reasonable to choose k larger than 30 on our benchmark data. While we do not present respective measurements here, we want to note that different choices of the weight values $w_1, \ldots, w_8$ hardly made any difference in the running times, neither did different metrics (in contrast to [11] where depending on the choice of the metrics, the speedup was reduced). In any case, for moderate values of k like $k = 24$ our scheme accelerates personalized route planning queries by one order of magnitude without incurring too much of a space overhead (for $k = 24$, the overlay graph is less than half the size of the original road network, queries are answered more than 13 times faster).

Very interesting in this context is how the running times and the speedups behave when adding more metrics. There are quite a few possible use cases for a large number of metrics. For example, one might induce a fine-grained partition of the roads of the network and then perform queries where

Table 11 Metrics used for our personalized route planning queries

Metric	Explanation
Travel time	Based on road categories
Eucl. dist	Simple euclidean distance
Height difference	Absolute value of height difference
Energy	Energy consumption
Edge type	OSM-edge type as a number
Speed	Maxspeed based on OSM tags
Rand	A random value
Unit	1 For each edge

Table 12 Personalized route planning queries on GER using the 8 metrics described in Table 11. A query consists of a randomly chosen source–target pairs and random weights $w_1, \ldots, w_8$. Measurements are averaged over 100 queries The speedup describes the factor between the total search time of our approach and the Dijkstra baseline

k	Dijkstra (ms)	Search local (ms)	Search overlay (ms)	Search total (ms)	Speedup
8	3282	0.01	481	481	6.82
12	3282	0.02	356	356	9.21
16	3282	0.03	295	295	11.1
20	3282	0.04	265	265	12.4
24	3282	0.04	249	249	13.1
28	3282	0.06	248	248	13.2

Table 13 Search times when increasing the number of metrics (random weights). BW graph, $k = 20$. Average of 100 random queries, speedup compared to plain Dijkstra

No. of metrics	Dijkstra (ms)	Total search (ms)	Speedup
2	338	27	12.5
4	352	29	12.1
8	377	35	10.8
16	419	41	10.2
32	521	55	9.5
64	654	80	8.2

certain classes of roads are disabled. This can be achieved by creating a metric for each road class which bears cost ∞ for the edges of the respective road class, 0 for the others. In a query, one can then disable a certain road class by choosing a multiplier of 1 for the respective metric. Another interesting scenario exists, which makes sense for rather short, e.g., commuter route planning queries. Here one can simulate time-dependent edge costs (longer travel times during rush hour) by associating different travel times on the edges for each hour of the day as a separate metric each. Again by choosing appropriate multipliers, one can perform the query on the respective road network at that time of the day. (Of course this only makes sense for rather short queries since we cannot express dynamic changes of edge costs over time in one route.)

In Table 13, we measured the behavior for a growing number of metrics for the smaller BW graph (we used k=20 which resulted in a cover of about 5 % the size of the original node set—computable in >20 s).

It turns out that even for quite a large number of metrics, the speedup compared to plain Dijkstra still is considerable and almost one order of magnitude. The absolute query time, of course, increases with growing number of metrics due to the more expensive evaluation of edge costs, but that is true for both the Dijkstra baseline and our approach.

7.7 Other network instances

Finally, we evaluate how our Pruning algorithm performs on social (and related) networks which differ significantly from road networks in terms of connectivity characteristics (much higher average and maximum node degree, smaller diameter).

We first compare our results for the k-SPC construction on the benchmark data set used in [17]. The cover sizes when applying adaptive sampling (AS)[18], the Set Cover-based approach (SC) [17] and Pruning are collected in Table 14 (the results for AS and SC are taken from [17]). We make the following observations: Pruning outperforms Adaptive Sampling on every considered instance, but for small k, the effect is less pronounced. SC produces better results than pruning on the CA-GrQc, CA-HepTh and Wiki-Vote network. On the other hand, pruning is superior to SC on all peer-to-peer network instances which might coincide with the smaller average degree in those networks compared to the other three ones. Also, pruning is much faster than SC. In [17] runtimes of several hours are reported already for $\epsilon = 5$. With those high run times, the applicability of SC is limited to small values of k and small networks. We could compute all results in Table 14 within 5 s per instance using Pruning.

To show that the Pruning approach is capable of dealing with even larger instances, we performed additional experiments on another benchmark set. Table 15 summarizes the results for computing k-APC and k-SPC in such networks.

The k-APC construction is remarkably fast. Even on these large networks (up to about 70 million edges), covers for $k = 5$ and $k = 10$ can be computed in >2 min. Also the compression is good. While the LiveJournal1 network originally contains over 4.8 million nodes, the cover for $k = 5$ contains >1.5 million nodes. But investigating the result for $k = 10$, we observe that the compression factor does not increase much. As the diameter of this network is 16, choosing k even larger makes little sense. Both the Amazon and Google networks exhibit a diameter of 21. Here, $k = 10$ leads to significantly smaller covers than $k = 5$. For all considered instances, the computed cover solutions were never more than a factor of 6 away from the lower bound.

Considering only shortest paths (see the lower half of Table 15), the compression naturally improves for all instances. But the runtime increases significantly, as especially for $k = 10$ large portions of the network have to

Table 14 Comparison of Adaptive Sampling (AS), the Set Cover-based approach (SC) and Pruning on a variety of networks. The ϵ-value refers to the notation used in [17]

	$k = 2$ ($\epsilon = 3$)		
	AS	SC	Pruning
CA-GrQc	2836	**896**	2773
CA-HepTh	5131	**2208**	4989
Wiki-Vote	2564	**1598**	2292
P2PG08	2359	2340	**2256**
P2PG09	2930	2904	**2865**
P2PG30	9688	9627	**9309**
P2PG31	16,493	16,394	**15,745**
	$k = 5$ ($\epsilon = 6$)		
	AS	SC	Pruning
CA-GrQc	908	**500**	656
CA-HepTh	2134	**1157**	1629
Wiki-Vote	2964	**571**	749
P2PG08	2043	1095	**861**
P2PG09	2256	1530	**1201**
P2PG30	8914	6845	**3922**
P2PG31	14,895	11,738	**6429**

Bold values indicate the best (minimal) solution among the three tested approaches (AS, SC, Pruning) per instance

Table 15 Experimental results for applying Pruning to large product, web, community and social networks. Timings are provided in seconds (s) or hours (h). For k-SPC, we used Pruning with minimality guarantee on Amazon0601 and web-Google, and Quick Pruning on com-lj

k-APC	$k = 5$		$k = 10$	
	$\|C\|$	Time (s)	$\|C\|$	Time (s)
Amazon0601	185,588	<1	142,801	20
web-Google	201,437	2	106,396	71
com-lj	1,058,952	8	715,303	13
LiveJournal1	1,395,008	12	1,142,169	63
k-SPC	$k = 5$		$k = 10$	
	$\|C\|$	Time	$\|C\|$	Time (h)
Amazon0601	143,107	50 s	82,708	1.6
web-Google	79,552	53 s	21,984	3.3
com-lj	872,118	5.5 h	552,323	22.2

be investigated in the Dijkstra runs. For the LiveJournal1 networks, computation times exceeded 24 hours and are therefore not reported. Still, using Quick Pruning, it is possible to compute a k-SPC on a social network with about 4 million nodes and 34 million edges (com-lj) in a few hours—an instance clearly intractable for the SC approach. Also note that pruning is an anytime algorithm. So we can just fix a time budget, stop the pruning process after this time budget exceeded, and have a valid solution at hand. Of course, the larger the time budget, the better the solution quality expectedly gets.

8 Conclusions

We introduced the k-All-Path Cover optimization problem with the goal of computing compact yet faithful synopses of the vertex set of road networks. Our proposed Pruning algorithm provides close-to-optimal results in practice and was experimentally proven to be very efficient on large graphs. For the special subcase of covering all k-node shortest paths as proposed by Tao et al. [18], we considerably improved their results in terms of running time and solution quality. But even for covering all paths consisting of k nodes, we could construct surprisingly small cover sets in reasonable time for moderate values of k.

On that basis, we developed a completely new framework for answering personalized route planning queries, where the user provides not only source and target but also weights for a given set of metrics. Our solution is based on a metric-independent overlay graph constructed upon our cover set in a preprocessing phase. While other route planners require a relatively expensive customization phase to adapt to personalized metrics, our approach allows to incorporate them in the overlay graph on the fly. This leads to a speedup of an order of magnitude compared to Dijkstra's algorithm. While this already allows for real-time query answering, a natural direction for future research is to aim for query times in the order of milliseconds as achievable for fixed metric shortest path queries. We want to emphasize that our speedup technique is somewhat orthogonal to speedup techniques like A^* and may be well combined with them. This might be a good starting point to achieve query times in the milliseconds range even for personalized route queries.

References

1. Abraham, I., Delling, D., Fiat, A., Goldberg, A.V., Werneck, R.F.: VC-dimension and shortest path algorithms. In: International Colloquium on Automata, Languages, and Programming (ICALP), pp. 690–699. Springer, Berlin (2011)
2. Abraham, I., Delling, D., Goldberg, A.V., Werneck, R.F.: A hub-based labeling algorithm for shortest paths in road networks. In: Symposium on Experimental Algorithms (SEA), pp. 230–241. Springer, Berlin (2011)
3. Bast, H., Funke, S., Sanders, P., Schultes, D.: Fast routing in road networks using transit nodes. Science **316**(5824), 566 (2007)
4. Brešar, B., Kardoš, F., Katrenič, J., Semanišin, G.: Minimum k-path vertex cover. Discrete Appl. Math. **159**(12), 1189–1195 (2011)
5. Brönnimann, H., Goodrich, M.T.: Almost optimal set covers in finite VC-dimension. Discrete Comput. Geom. **14**(1), 463–479 (1995)

6. Delling, D., Goldberg, A.V., Pajor, T., Werneck, R.F.F.: Customizable route planning. In: Symposium on Experimental Algorithms (SEA), pp. 376–387. Springer, Berlin (2011)
7. Delling, D., Kobitzsch, M., Werneck, R.: Customizing driving directions with GPUs. In: Euro-Par Parallel Processing, vol. 8632, pp. 728–739. Springer, Berlin (2014)
8. Dibbelt, J., Pajor, T., Wagner, D.: User-constrained multi-modal route planning. Networks **6**, 10 (2012)
9. Dibbelt, J., Strasser, B., Wagner, D.: Customizable contraction hierarchies. arXiv preprint arXiv:1402.0402 (2014)
10. Funke, S., Nusser, A., Storandt, S.: On k-path covers and their applications. PVLDB **7**(10), 893–902 (2014). http://www.vldb.org/pvldb/vol7/p893-funke.pdf
11. Funke, S., Storandt, S.: Polynomial-time construction of contraction hierarchies for multi-criteria objectives. In: Sanders, P., Norbert, Z. (eds) Algorithm Engineering and Experiments (ALENEX), pp. 41–54. SIAM (2013)
12. Geisberger, R., Sanders, P., Schultes, D., Vetter, C.: Exact routing in large road networks using contraction hierarchies. Transp. Sci. **46**(3), 388–404 (2012)
13. Gutman, R.J.: Reach-based routing: a new approach to shortest path algorithms optimized for road networks. In: Lars, A., Giuseppe, F.I., Robert, S. (eds) Algorithm Engineering and Experiments (ALENEX), pp. 100–111 (2004)
14. Haussler, D., Welzl, E.: Epsilon-nets and simplex range queries. In: Symposium on Computational Geometry (SCG), pp. 61–71. ACM (1986)
15. Khot, S.: On the power of unique 2-prover 1-round games. In: Proceedings of the Thiry-fourth Annual ACM Symposium on Theory of Computing. STOC '02, pp. 767–775. ACM, New York, NY (2002)
16. Khot, S., Regev, O.: Vertex cover might be hard to approximate to within 2- ε. J. Comput. Syst. Sci. **74**(3), 335–349 (2008)
17. Ruan, N., Jin, R., Huang, Y.: Distance preserving graph simplification. In: Data Mining (ICDM), 2011 IEEE 11th International Conference on, pp. 1200–1205. IEEE (2011)
18. Tao, Y., Sheng, C., Pei, J.: On k-skip shortest paths. In: ACM SIGMOD International Conference on Management of Data, pp. 421–432. ACM (2011)
19. Vapnik, V.N., Chervonenkis, A.Y.: On the uniform convergence of relative frequencies of events to their probabilities. Theory Probab. Appl. **16**(2), 264–280 (1971)