On k-Path Covers and their Applications

Stefan Funke
Universität Stuttgart
Germany

funke@fmi.uni-stuttgart.de

André Nusser
Universität Stuttgart
Germany

nusser@fmi.uni-stuttgart.de

Sabine Storandt
Universität Freiburg
Germany

storandt@cs.uni-freiburg.de

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* path 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 theoretical 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).

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 lead 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 less than 30 million nodes whereas in 2013 this number has grown to more than 2 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 first look at a few applications for SNDBs which can benefit from small k-Path Covers.

---

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

Application 1: 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 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, of course, 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. Similarly, if the goal is to acquire statistics along paths (like 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 both in terms of space requirement as well as running time for a single query.

Application 2: 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 would be 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 Figure 1.

Application 3: 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, different vehicles exhibit different turn costs and so on. So a query might not only consist 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
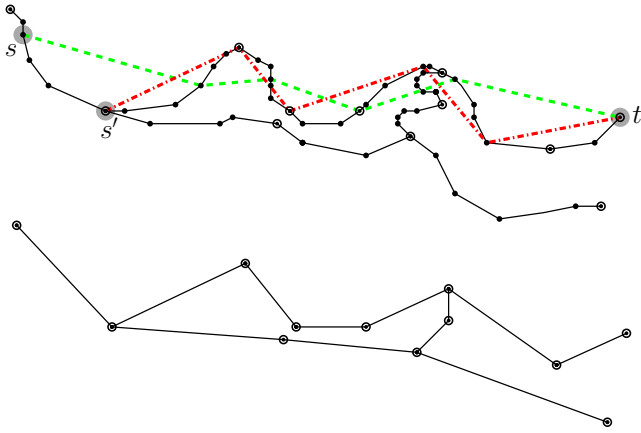
Figure 1: Small hiking map. In the upper image the dashed red and green line indicate two hiking routes ($s \to t$ and $s' \to t$) simplified greedily using $k = 6$. While $s' \to t$ is a subroute of $s \to t$, the two simplified routes do not resemble each other at all. The circled nodes are a feasible global cover for $k = 6$, resulting in a nice simplification of the map presented in the lower image.

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.

Related Work

Tao et al. in [12] 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 speed-up techniques like *reach* [9]). We want to note, though, that the achievable query times have meanwhile been superseded by current speed-up techniques like *Transit Nodes* [3], *Contraction Hierarchies (CH)* [8] or *Hub Labels* [2] which answer queries faster by several orders of magnitude. A fundamental restriction of all these techniques (including [12]) 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 some 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. But as soon as the metric changes, phase two has to be redone. For the CRP framework, the metric customization takes several seconds (on a rather powerful multicore machine). While this is tolerable for simulating traffic dependent edge costs or when customizing for a specific *single* user, some seconds are too slow when every query demands its own metric (in fact running plain Dijkstra might then be faster). We will take care of such highly dynamic queries by also applying a metric-independent preprocessing at 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 [7] the authors adapt the CH preprocessing technique to the case where each edge e in the network has d associated costs $c_1(e), c_2(e), \ldots, c_d(e)$. A query consists of source, destination and non-negative multipliers $\alpha_1, \alpha_2, \ldots, \alpha_d$; the data structure returns the shortest paths under edge costs $c(e) = \alpha_1 c_1(e) + \alpha_2 c_2(e) + \cdots + \alpha_d c_d(e)$. Experimental results proved the technique to yield high speed-ups for $d = 2$ and 3. Still, the speed-up for $d = 3$ was already considerably lower than for $d = 2$, 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. $d > 4$) of unsimilar costs this approach comes at its limits.

Our Results

The main contributions of our paper are the following:

- We generalize the work of [12] 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 [12] 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 properties (covering *shortest k-paths*) as the ones derived in [12]; 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 [12]).

- More on the theoretical 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.

Outline

After providing formal definitions in Section 2, we review in Section 3 theoretical results on the complexity of the k-Path Cover problem, and the k-Shortest-Path Cover problem respectively. As k-Path Cover is NP-hard [4] and k-Shortest-Path Cover turns out to be NP-hard as well, we investigate approximation algorithms based on low VC-dimension of the underlying set system. In Section 4 we develop practical algorithms to construct k-All-Path Covers and k-Shortest-Path Covers as well as instance-based lower bounds. We

conclude with an experimental section showing the practicability of our developed algorithms and in particular looking at the use case of personalized route planning.

2. FORMAL DEFINITIONS

To formalize our cover problems, we introduce the following notations: The input is a simple directed graph $G(V, E)$. Therefore a path π in G can be uniquely represented as the list of its traversed vertices. Throughout this paper, when using the term 'path' we always refer to simple paths, that is, no vertex appears more than once. 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 (di)graph $G(V, E)$ and $k \in \mathbb{N}$, select a minimum subset of vertices $C \subseteq V$ such that for every simple path $\pi = v_1, \cdots, v_k$ 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 lead 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 (di)graph $G(V, E, c)$ and $k \in \mathbb{N}$, select a minimum subset of vertices $C \subseteq V$ such that for every shortest (according to c) path $\pi = v_1, \cdots, v_k$ 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 [12] 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 $c_1, c_2, \cdots, c_r : E \to \mathbb{R}^+$ 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 $w_1, w_2, \ldots, w_r \in \mathbb{R}^+$). 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. THEORETICAL ANALYSIS

In this section we give a brief overview of complexity results for k-APC and k-SPC. Then we derive a $\log|OPT|$-approximation for k-SPC based on VC-dimension analysis.

3.1 NP-hardness and Approximation

In [11], the MinimumGateVertexSet (MGS) problem was analyzed, which is closely related to k-SPC. Here, the authors proved NP-hardness for a generalized problem version,

namely, in terms of our setting, a variant where only a pre-specified 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 theoretical 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 holds, their results even imply APX-hardness for $2 - \epsilon$. Furthermore we would like to point out that the APX-hardness result carries over to k-SPC, because augmenting all edges with uniform costs in the reduction gadget used in [4] makes all paths of length k shortest paths. Hence, their proposed k-approximation based on converting the problem to an instance of HittingSet (as defined for the sake of notation in the following) is also valid for k-SPC.

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 (shortest) path of length k as set of its contained vertices to construct the HittingSet instance. So for k-APC and k-SPC every set has exactly k elements, immediately implying a valid k-approximation.

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 [12]. 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 [12], 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 [10] we can find a k-SPC of size $\mathcal{O}(d(n/k) \log(n/k))$ with d being the VC-dimension of the set system [13].

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

So in the setting of S being a set of paths, the VC-dimension describes an exclusive upper bound on how often two paths can intersect in a non-contiguous manner. Obviously if shortest paths are ambiguous in G, the VC-dimension cannot be bounded. But as already shown in [11], 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 speed-up techniques) and [12] 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 [12] that the result is valid for general graphs, also including directed graphs. This is not true, as the example in

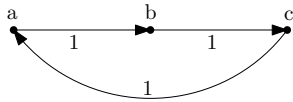Figure 2 shows. There a directed path of three nodes indeed can be shattered disproving 2 as an upper bound.

Figure 2: 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.

But as dealing with *directed* edges is naturally required when considering street graphs (asymmetric edge weights, one-way streets, roundabouts), 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 (UD-SPS) 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.

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 $\{v_1, v_2, v_3, v_4\}$ cannot be shattered. If there exists no shortest path containing all the nodes $v_1, \cdots, v_4$ 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 $B_1 = \{v_1, v_4\}$, $B_2 = \{v_2, v_4\}$ and $B_3 = \{v_1, v_2, v_4\}$ and assume they can be realized by paths $q_1, q_2, q_3 \in A$. Because q_1 does not contain v_3 (and v_2) and shortest paths are unique, it follows $q_1 = \cdots, v_4, \cdots, v_1, \cdots$. With the same argument B_2 leads to $q_2 = \cdots, v_4, \cdots, v_2, \cdots$. For B_3 observe, that for all ordered pairs of its elements besides (v_2, v_1) the shortest path is already known and none of them is consistent with q_3. But a path from v_2 to v_1 over v_4 must contain v_3. Hence there exists no q_3 and $\{v_1, v_2, v_3, v_4\}$ 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 the following we will develop approaches for efficient cover set construction in practice. We will first discuss the general case of k-Path Covers but then also consider 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 retrieve a feasible cover C. Unfortunately, this is not practical for large input graphs, as the exploration time as well as the space consumption 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 ingoing paths of v until reaching other nodes from C. If the combination of such an outgoing and ingoing path 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. To decide whether a node v can be pruned, the procedure

---

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 $\exists \pi \in P_o$ with $|\pi| = k$ **then**
    return true
end if
for all $\pi \in P_o$ **do**
    search for the longest incoming path into v
        not containing nodes in $(C \cup \pi) - \{v\}$
    if such path of length $k - |\pi| + 1$ exists **then**
        return true
    end if
end for
return false

---

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). While these subroutines 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. Also observe that both the construction of the set P_o as well as the search for an incoming path will only explore paths of length at most k since by assumption C was a valid cover before consideration of node v. And in particular at the very beginning, when C is almost the whole vertex set, these two steps abort almost immediately. This is one reason for the pruning approach to be much faster than the naive algorithm 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. Therefore no node can be removed from the final C without invalidating the solution. Hence C has to be minimal in a set theoretic sense. $\square$

Of course, minimality of C does not imply that it is also minimum, i.e. of minimal cardinality amongst all possible covers.

Observe that the order in which nodes are considered during the course of the algorithm highly influences the solution quality (while not for feasibility or minimality). Intuitively, nodes with a low degree might not cover as many paths as nodes embedded in dense clusters. We will provide an experimental study measuring the influence of different node order schemes for pruning towards the end of the paper.

4.2 Lower Bounds

Unfortunately, there are no meaningful theoretical 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 non-intersecting 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 Figure 3 a small illustration for the lower bounding technique is provided.
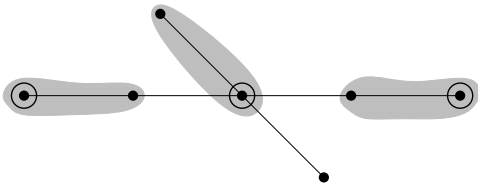

Figure 3: The set of circled nodes shows a feasible 2-APC. The highlighted paths are disjoint and therefore prove a lower bound of 3 on the size of a feasible cover for this instance.

4.3 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 $C_1, \cdots, C_r$ 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.

4.4 Overlay-Graph Construction

For several application scenarios we not only require the cover set C but also the overlay graph induced by this set. That means, for any two nodes $v, w \in C$ for whom there exists a path in G from v to w containing no other nodes from C, we create an edge (v, w) in the overlay graph. We denote the resulting structure by $G_O(C, E_O)$. To construct G_O, we proceed as follows: We run breadth first search on G (BFS) from every 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).

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 [12]. 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 keep us from inserting superfluous nodes. 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).

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[2] 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 $k_F + k_R - 1 > k$, add v back to C otherwise prune it.

---

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

We call this algorithm *Quick Pruning* because it runs very fast in practice. 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. 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 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. Therefore we expect the quality to be superior to *Quick Pruning* but the runtime to be worse. So it depends on the application context which approach to use.

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 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 $c' : E \to \mathbb{R}$ 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). 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 amongst these can easily be determined for all nodes in D in $O(|D|)$ time.

	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 1: Benchmark graphs ($M = 10^6$).

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. EXPERIMENTAL EVALUATION

5.1 Environment and Data Sets

Our C++ implementations were compiled using gcc 4.6.3 and benchmarked on a 3.2GHz intel i5-3470 with 16GB RAM. As benchmark data we used road networks extracted from the OpenStreetMap (OSM) project as well as the standard DIMACS road network graphs[3], which were also used in [12]. 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 4.5 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.

5.2 Constructing k-APC

Let us start with the pruning approach for constructing sets C covering *all* paths consisting of k nodes: In Table 2 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`) as well as 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`. `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 as well as lower bounds. Note that our instance-based 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 3 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

---

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

G	order	lb	secs	$|C|$	rel.	secs
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

Table 2: k-**APC: Influence of different node orders on cover size and lower bounds (lb) for** $k = 16$**. The columns are from left to right: graph, order, lower bound size, lower bound construction time, cover size, relative cover size, cover construction time.**

G	k	lb	$|C|$	perc	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 3: k-**APC: Approximation ratios (apx) and construction times for comp-inc order and varying values of** k**. 'perc' describes the fraction of nodes in** V **that are contained in the cover** C**.**

be much closer to our constructed covers than to our rather naive lower bound.

5.3 Special case: Constructing k-SPC

For comparison with the results in [12] 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 4. The outcomes of our implementation of the Adaptive Sampling approach are pretty close to their reported performances (maybe 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 ([12] 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. Running times are slightly above the ones for Adaptive Sampling yet our Quick Pruning variant (which does not guarantee minimality) was always much faster than adaptive sampling but still better in terms of quality of the solution.

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

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 [12]) 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).
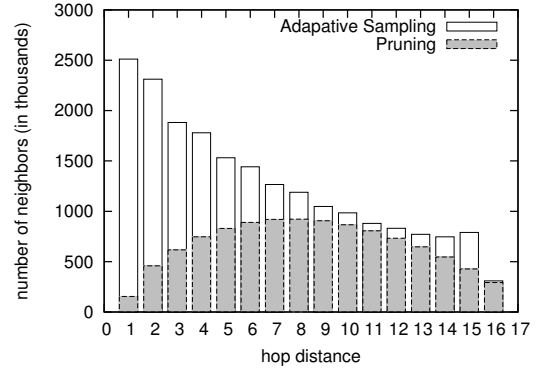

Figure 4: Distribution of hop distances between neighboring nodes in the USA skip-graph for $k = 16$**.**

In addition, we analyzed the hop distances between neighboring nodes for Adaptive Sampling (AS) and Pruning, exemplary for USA with $k = 16$. Here two nodes in C are called neighbors if the shortest path between them in G contains no other nodes from C. Obviously large hop distances between neighbors is a quality indicator for C. 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 Figure 4. 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 have a larger hop distance.

Considering the representation of paths subsampled according to C, we can compare the number of edges of this simplified path to the number of original edges in G to measure the compression. The resulting numbers can be found for varying values of k in Table 5. 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.

5.4 Application: Personalized Route Planning

Let us now instrument k-All Path Covers to speed-up fully personalized route planning queries. As input we are given a

G	k	lower bound	Adaptive Sampling	time (secs)	Quick Pruning	time (secs)	Pruning	time (secs)
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	2,700	1,256,871	513	584,904	4,970
	40	136,592	1,564,624	4,401	1,004,268	772	431,686	9,520
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	1,179	648,508	322	355,062	1,360
	48	135,545	925,359	1,584	530,949	433	285,780	2,050
	56	107,687	825,694	2,214	447,816	546	237,479	2,900

Table 4: k-SPC: Sampling vs. Pruning: Comparison for USA and GER.

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%

Table 5: Ratio of edges on a k-sampled path compared to all edges in the original path. Values are averaged over 1000 random queries.

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 [7], which is only practicable for a small number of metrics ($r \leq 3$), we are not aware of any speed-up 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 seconds on a multicore server, 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 and works as follows:

- Preprocessing:

  - compute a k-APC C
  - construct the overlay *multigraph* $G_O(C, E_O)$ wrt $G(V, E)$, where there is an edge $(v, w) \in E_O$ for every path π in G between nodes $v, w \in C$ which does not contain other nodes of C. The r cost values of an edge $e \in E_O$ arise from component-wise addition of the cost values of the edges of the respective path.

- Query:

  - 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 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 multigraph $G_O(C, E_O)$ 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. At this point 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 $d(s, a_t) + d(a_t, t)$.

The efficiency of our approach is based on the fact that the search in the overlay multigraph $G_O(C, E_O)$ turns out to be much faster than on the original graph $G(V, E)$ and the initial searches for $A(s)$ and $A(t)$ take negligible time. Let us first experimentally examine the cost and structure of the overlay multigraph $G_O(C, E_O)$.

5.4.1 Personalized Route Planning: Preprocessing

In Table 6 we see the characteristics of the overlay multigraphs constructed for our benchmark graphs for different values of k. It is not surprising that the number of edges in the overlay multigraph is much higher than in a respective *shortest-path* overlay multigraph (where there is at most one edge between any pair of nodes) since every path between two nodes in the cover needs to be represented by a respective edge. While the number of nodes in the cover decreases rapidly with growing value of k, the number of edges in the overlay multigraph first remains almost constant but then increases quickly, surpassing the number of edges of the original graph for large values of k. 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.

	k	$|C|$	$|E_O|$	time (sec)	total (sec)	avg. deg	max deg
BW	8	264,662	1,103,606	4.4	7.6	4	99
	12	187,398	1,091,776	4.4	8.5	5	236
	16	146,761	1,195,321	5.1	10.1	8	525
	20	121,750	1,389,930	6.2	12.4	11	1,313
	24	104,899	1,750,495	8.1	15.9	16	5,470
	28	92,428	2,374,019	11.7	21.1	25	38,675
	32	83,173	3,036,534	15.1	27.1	36	14,351
	36	75,977	4,421,291	23.5	39.4	58	65,910
	40	70,185	7,677,834	45.1	68.7	109	278,315
GER	8	2,308,934	9,117,057	35.6	65.6	3	116
	16	1,209,215	9,504,005	45.5	92.1	7	1,589
	24	845,905	15,050,313	79.0	149.4	17	30,539
	32	666,829	32,057,249	186.8	304.6	48	163,654

Table 6: Personalized Route Planning/Preprocessing. Construction of k-APC and overlay graph: size of k-APC, number of edges in the overlay graph, time to construct the overlay graph, total time (k-APC and overlay graph construction), avg. and maximum degree in overlay graph.

k	Dijkstra (ms)	search local (ms)	search overlay (ms)	search total (ms)	speed-up
8	3,282	0.01	481	481	6.82
12	3,282	0.02	356	356	9.21
16	3,282	0.03	295	295	11.1
20	3,282	0.04	265	265	12.4
24	3,282	0.04	249	249	13.1
28	3,282	0.06	248	248	13.2

Table 7: Personalized Route Planning Queries on GER: 8 metrics, random source-target pairs, random weights $w_1, \ldots, w_8$. Averages for 100 random queries: Dijkstra baseline, search for access nodes, search in overlay multigraph, total search time, speed-up vs. Dijkstra.

5.4.2 Query Processing

We considered the following 8 metrics for personalized route planning:

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

For query evaluation we generated 100 random source-target queries with random weights $w_1, w_2, \ldots, w_8$ comparing the performance of our speed-up scheme for different values of k with a standard Dijkstra run. Apart from the measured query times, we will 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 7 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, the search in the overlay multigraph clearly dom-

inates the overall query time. The achieved speed-up first quickly grows with increasing k but improves only slightly above $k = 20$. One should bear in mind, though, that the overlay multigraph gets very dense for large k (see Table 6), 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 [7] where depending on the choice of the metrics, the speed-up was greatly 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 multigraph 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 speed-ups 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 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 8 we measured the behavior for a growing number of metrics for the smaller BW graph.

It turns out that even for quite a large number of metrics, the speed-up compared to plain Dijkstra still is considerable and almost one order of magnitude. The absolute query

# of metrics	Dijkstra (ms)	total search (ms)	speed-up
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

Table 8: Search times when increasing the number of metrics (random weights). BW graph, $k = 20$. Average of 100 random queries, speed-up compared to plain Dijkstra.

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 as well as our approach.

6. CONCLUSION

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. [12], 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 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 multigraph 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 speed-up 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 speed-up techniqe is somewhat orthogonal to speed-up techniques like A^* and may be very well combined with them. This might be a good starting point to achieve query times in the milliseconds range even for personalized route queries.

7. REFERENCES

[1] I. Abraham, D. Delling, A. Fiat, A. V. Goldberg, and R. F. Werneck. Vc-dimension and shortest path algorithms. In *International Colloquium on Automata, Languages, and Programming (ICALP)*, pages 690–699. Springer, 2011.

[2] I. Abraham, D. Delling, A. V. Goldberg, and R. F. Werneck. A hub-based labeling algorithm for shortest paths in road networks. In *Symposium on Experimental Algorithms (SEA)*, pages 230–241. Springer, 2011.

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

[4] B. Brešar, F. Kardoš, J. Katrenič, and G. Semanišin. Minimum k-path vertex cover. *Discrete Applied Mathematics*, 159(12):1189–1195, 2011.

[5] H. Brönnimann and M. T. Goodrich. Almost optimal set covers in finite vc-dimension. *Discrete & Computational Geometry*, 14(1):463–479, 1995.

[6] D. Delling, A. V. Goldberg, T. Pajor, and R. F. F. Werneck. Customizable route planning. In P. M. Pardalos and S. Rebennack, editors, *Symposium on Experimental Algorithms (SEA)*, pages 376–387. Springer, 2011.

[7] S. Funke and S. Storandt. Polynomial-time construction of contraction hierarchies for multi-criteria objectives. In *Algorithm Engineering and Experiments (ALENEX)*, pages 41–54. SIAM, 2013.

[8] R. Geisberger, P. Sanders, D. Schultes, and C. Vetter. Exact routing in large road networks using contraction hierarchies. *Transportation Science*, 46(3):388–404, 2012.

[9] R. J. Gutman. Reach-based routing: A new approach to shortest path algorithms optimized for road networks. In *Algorithm Engineering and Experiments (ALENEX)*, pages 100–111, 2004.

[10] D. Haussler and E. Welzl. Epsilon-nets and simplex range queries. In *Symposium on Computational Geometry (SCG)*, pages 61–71, New York, NY, USA, 1986. ACM.

[11] N. Ruan, R. Jin, and Y. Huang. Distance preserving graph simplification. *CoRR*, abs/1110.0517, 2011.

[12] Y. Tao, C. Sheng, and J. Pei. On k-skip shortest paths. In *ACM SIGMOD*, pages 421–432. ACM, 2011.

[13] V. N. Vapnik and A. Y. Chervonenkis. On the uniform convergence of relative frequencies of events to their probabilities. *Theory of Probability and its Applications*, 16(2):264–280, 1971.