A Sidetrack-Based Algorithm for Finding the k Shortest Simple Paths in a Directed Graph*

Denis Kurz[1] and Petra Mutzel[2]

1 **Department of Computer Science, TU Dortmund, Germany**
`denis.kurz@tu-dortmund.de`
2 **Department of Computer Science, TU Dortmund, Germany**
`petra.mutzel@tu-dortmund.de`

Abstract

We present an algorithm for the k shortest simple path problem on weighted directed graphs (kSSP) that is based on Eppstein's algorithm for a similar problem in which paths are allowed to contain cycles. In contrast to most other algorithms for kSSP, ours is not based on Yen's algorithm [19] and does not solve replacement path problems. Its worst-case running time is on par with state-of-the-art algorithms for kSSP. Using our algorithm, one may find $\mathcal{O}(m)$ simple paths with a single shortest path tree computation and $\mathcal{O}(n+m)$ additional time per path in well-behaved cases, where n is the number of nodes and m is the number of edges. Our computational results show that on random graphs and large road networks, these well-behaved cases are quite common and our algorithm is faster than existing algorithms by an order of magnitude.

1998 ACM Subject Classification G.2.2 Graph Theory

Keywords and phrases directed graph, k-best, shortest path, simple path, weighted graph

Digital Object Identifier 10.4230/LIPIcs.ISAAC.2016.49

1 Introduction

The k *shortest path problem* in weighted, directed graphs (kSP) asks for a set of k paths from a *source* s to a target *target* t in a graph with n nodes and m edges. Every path that is not output by an algorithm should be at least as long as any path in the output. Algorithms for this problem can be useful tools when it is hard to specify constraints that a solution should satisfy. Instead of computing only one shortest path, kSP algorithms generate k paths, and the user can then pick the one that suits their needs best. The best known algorithm for this problem runs in time $\mathcal{O}(m + n \log n + k \log k)$ and is due to Eppstein [4]. In the initialization phase, the algorithm builds a data structure that contains information about all s-t paths and how they interrelate with each other, in time $\mathcal{O}(m + n \log n)$. This can be reduced to $\mathcal{O}(m + n)$ if the shortest path tree (SP tree) can be computed in time $\mathcal{O}(m + n)$. In the enumeration phase, a *path graph* is constructed. The path graph is a quaternary min-heap where every path starting in the root correlates to an s-t path in the original graph. We require $\mathcal{O}(k \log k)$ time for the enumeration phase if we want the output paths to be ordered by length. If the order in which the paths are output does not matter, Frederickson's heap selection algorithm [7] can be used to enumerate the paths after the initialization phase in time $\mathcal{O}(k)$.

The k *shortest simple path problem* (kSSP), introduced in 1963 by Clarke, Krikorian and Schwartz [2], seems to be more expensive, computationally. In contrast to kSP, the computed

* This work was partially supported by the German Research Foundation, RTG 1855.

paths are required to be simple, i.e., they must not contain a cycle. The extra effort may be well-invested if many of the k shortest paths are non-simple and we are only interested in simple paths. The algorithm by Yen [19] used to have the best theoretical worst-case running time of $\mathcal{O}(kn(m + n \log n))$ for quite some time. Gotthilf and Lewenstein [9] improved upon this bound recently. They observed that kSSP can be solved by solving $\mathcal{O}(k)$ all pairs shortest path (APSP) instances. Using the APSP algorithm by Pettie [13], they obtain a new upper bound of $\mathcal{O}(kn(m + n \log \log n))$. Vassilevska Williams and Williams [18] showed that, for constant k, an algorithm for kSSP with running time $\mathcal{O}(n^{3-\varepsilon})$ for some positive ε (*truly subcubic*) would also yield algorithms with truly subcubic running times for some other problems, including APSP. A recent survey of the field is due to Eppstein [5]. The kSSP on undirected graphs seems to be significantly easier. Katoh et al. [11] proposed an algorithm that solves kSSP on undirected graphs in time $\mathcal{O}(k(m + n \log n))$.

A subproblem occurring in Yen's algorithm is the (restricted) *replacement path problem*. Given a shortest s-t path p in a graph, it asks for a set of paths as follows. For each $i < |p|$, the set has to include a shortest simple path that uses the first $i - 1$ edges of p, but not the ith. This problem has to be solved $\mathcal{O}(k)$ times to find the k shortest simple paths using Yen's algorithm. In the original version of Yen's algorithm, the replacement paths are found using $\mathcal{O}(|p|)$ shortest path computations, resulting in time $\mathcal{O}(n(m + n \log n))$. Hershberger et al. [10] compute one SP tree rooted in s and one reversed SP tree rooted in t, respectively. They use these two trees to find a replacement path in constant time per edge on p, cutting down the time required to find all replacement paths to $\mathcal{O}(m + n \log n)$ when Dijkstra's algorithm is used. However, the paths generated this way are not guaranteed to be simple. Such non-simple paths can be detected in constant time and repaired by falling back to Yen's replacement path computation for the path edge in question. Since they do not provide an upper bound for the number of non-simple paths that may occur using this method, the worst-case running time is again $\mathcal{O}(n(m + n \log n))$.

Some approaches reuse one fixed reversed SP tree T_0 rooted in t and computed during the initialization of their kSSP algorithm, in contrast to $\mathcal{O}(1)$ SP trees per replacement path instance. Pascoal [12] noticed that the replacement path that deviates from p at node v might be one that uses an edge (v, w) to an unused successor w of v and then follows the path from w to t in T_0. Therefore, they test whether the shortest such path is simple, and fall back to a full shortest path computation if it is not. Although they do not describe in detail how this check is done, it can be done in time $\mathcal{O}(m + n)$ per replacement path instance by partitioning the nodes into blocks as described by Hershberger et al. [10]. Feng [6] uses the reversed SP tree to partition V into three classes. For each edge (u, v) on p for which we want to compute a replacement path, *red* nodes have already been used to reach v via p. A *yellow* node v is a non-red upstream node of some red node in T_0, i.e., the path from v to t in T_0 contains a red node. All other nodes are *green*. They then do shortest path computations from v using Dijkstra's algorithm like Yen. However, they are able to restrict the search to yellow nodes, resulting in a significantly smaller search space. Feng does not provide upper bounds on the size of this search space, resulting again in a worst-case running time of $\mathcal{O}(n(m + n \log n))$ for each replacement path instance.

The fact that the upper time bound for exact kSSP algorithms has not been improved for a long time inspired research on inexact approaches. This line of research so far spans three publications that all use the notion of a *detour* of a path, which is the (connected) subpath of a replacement path that diverts from a reference path. It was started by Roditty and Zwick [15] who proposed a Monte Carlo $\mathcal{O}(m\sqrt{n} \log n)$ algorithm for the replacement path problem on directed graphs with small integer weights. They also proposed a framework that solves

kSSP by $\mathcal{O}(k)$ computations of second shortest simple paths in appropriate subgraphs, which in turn is solved by their replacement path algorithms. Roditty [14] enhanced this framework to allow for approximate kSSP algorithms when an approximation algorithm for the second shortest path subproblem is used. They also provided such an $\frac{3}{2}$-approximation algorithm, leading to a $\frac{3}{2}$-approximation algorithm for kSSP with running time in $\mathcal{O}(k\sqrt{n}(m+n\log n))$. An α-approximation algorithm guarantees that the i-th output path is at most α times as long as an actual i-th shortest path. The algorithm for approximate second shortest simple paths distinguishes between short and long detours. This approach was extended by Bernstein [1] from two to $\mathcal{O}(\log n)$ classes of detours, which are handled in increasing order. This way, they were able to obtain an algorithm that gives $(1+\varepsilon)$ approximations in $\mathcal{O}(\varepsilon^{-1}\log^2 n\log(nC/c)(m+n\log n)$ time, where c, C are the minimum and maximum edge cost, respectively, giving the first (approximation) algorithm that breaks the $\mathcal{O}(m\sqrt{n})$ barrier. See Frieder and Roditty [8] for an experimental study of Bernstein's algorithm.

Our contribution. We propose an algorithm that was derived from Eppstein's notion of a path graph [4]. Our algorithm achieves the same worst-case running time as Yen's algorithm. Like Yen, we rely on shortest path (tree) computations. In contrast to Yen-based algorithms, however, our algorithm may draw $\mathcal{O}(m)$ simple paths from one SP tree computation. If the underlying graph is acyclic, the revised algorithm at the end of this paper requires $\mathcal{O}(n\log n + k(m+n))$ without further modifications. Alternatively, one could test whether the graph is acyclic and then use Eppstein's algorithm. However, this method fails if the graph has just a single cycle, in which case our algorithm appears to be a good choice. As most other kSSP algorithms, our algorithm works on multigraphs without modification. After some definitions in Section 2, we propose a simplified version of our algorithm with running time $\mathcal{O}(km(m+n\log n))$ in Section 3. In Section 4, we show how this running time can be reduced to $\mathcal{O}(kn(m+n\log n))$, and how to improve the running time in practice even further. Finally, we present the results of our computational studies in Section 5 to prove the efficiency of our algorithm.

2 Definitions

Let $G=(V,E)$ be a directed graph with *node* set V and *edge* set E. Let $s,t\in V$ be *source* and *target* nodes, respectively. We assume an implicit edge weight function $c: E\to\mathbb{R}_0^+$ throughout this paper. We denote the number of nodes $|V|$ by n and the number of edges $|E|$ by m. A *path* connecting v to w in G, or v-w path, is an edge sequence $p=(e_1,e_2,\dots,e_r)$, $e_i=(v_i,w_i)$, with $v=v_1$, $w=w_r$ and $w_i=v_{i+1}$ for $1\le i<r$. For the sake of simplicity, we only consider combinations of G, s and t such that there exists an s-v path and a v-t path in G for every $v\in V$. A node u is said to be on the path p, denoted by $u\in p$, if $u=w$ or $u=v_i$ for some i. If $v_i\neq v_j\neq w$ for $1\le i,j\le r$, p is a *simple* path. The *prefix* $(e_1,\dots,e_i)$ is a v-w_i path and denoted by p^i. The *length* $c(p)$ of the path p is the sum of edge weights of its edges. If every v-w path is at least as long as p, it is called a *shortest v-w path*. We write $G-p$ to denote the induced subgraph $G[\{v\in V\mid v\notin p\}]$.

The *k shortest simple path problem* (kSSP) is an enumeration problem. Given a directed graph $G=(V,E)$ with source node $s\in V$, target node $t\in V$, edge weights c, and some $k\in\mathbb{N}$, we want to compute a set P comprising k simple paths from s to t in G such that $c(p)\le c(p')$ for every pair $p\in P$, $p'\notin P$ of simple paths. We obtain the *k shortest path problem* (kSP) if we do not require the computed paths to be simple.

A *shortest path tree* (SP tree) T of G is a subtree of G with node set V such that each $v\in V$ has exactly one outgoing edge, which lies on a shortest v-t path, or no outgoing edges

if no such edge exists. We denote the latter case by $v \notin T$. Our algorithm will compute several SP trees, the first of which we call *initial SP tree* T_0. An edge $e \notin T$ is called *sidetrack* w.r.t. T; we will omit T in most cases. For a sidetrack $e = (v, w)$, the *sidetrack cost* $\delta_T(e)$ is defined as $(c(e) + d(w)) - d(v)$, where $d(u)$ is the length of the unique u-t path in T. The sidetrack cost is therefore the difference between the length of a shortest v-t path and the length of a shortest v-t path that starts with e. The sidetrack set $D_T(v)$ of a node $v \in V$ is the set of all sidetracks w.r.t. T with tails on the unique v-t path in T. When sidetracks are organized in heaps, we use sidetrack costs for comparison.

Let $p = (e_1, \ldots, e_k)$, $p' = (f_1, \ldots, f_l)$ be two s-t paths, and $i^* = \max\{i \mid e_j = f_j \text{ for } 1 \leq j < i\}$. Then, with respect to p, i^* is the *deviation index*, the tail of e_{i^*} is the *deviation node* $\mathrm{dev}(p')$, and e_{i^*} is the *deviation edge* of p'. As is usual for kSSP algorithms, we will discover paths in a hierarchical fashion. We manage a *candidate set*, i.e., a set of *candidate paths* that have been found, but have not been determined to be one of the k shortest simple paths. Any path p that is *extracted* from the candidate set will be part of the solution; candidate paths that are found not to be part of the solution are *discarded*. A range of new candidate paths is derived from p. Any derived path p' is added to the candidate set. We call p the *parent path* of p'. When p is omitted, the terms deviation node and edge are w.r.t. the parent path of p'. By removing the deviation edge of p from p, p is split into its *prefix path* $\mathrm{pref}(p) := p^{i^*}$ starting in s, and its *suffix path* $\mathrm{suff}(p)$ ending in t. The initial s-t path p_0 in T_0 has no parent path and thus no deviation edge. We define its suffix path to be p_0 itself.

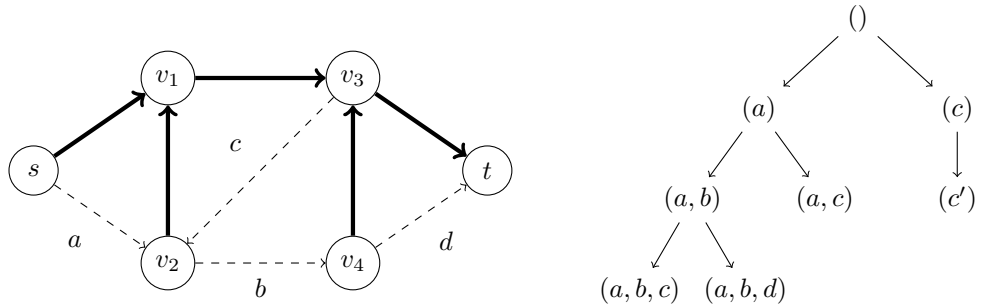
We generalize Eppstein's representation [4] for paths. Eppstein represents paths as sequences of sidetracks, all w.r.t. the same SP tree. In our representation, every sidetrack e in a sidetrack sequence may be associated with a different SP tree T_e. The path represented by a sidetrack sequence $(e_1, \ldots, e_r)$ can then be reconstructed as follows. Starting in s, we follow the initial SP tree T_0 until we reach the tail of e_1. After reaching the tail of e_i, we traverse e_i and follow T_{e_i} until we reach the tail of e_{i+1}, or, in case $i = r$, until we reach t. Note that Eppstein's representation is the special case where $T_e = T_0$ for each e in a sidetrack sequence, and that both Eppstein's sidetrack sequences and our generalized ones may represent non-simple paths. The distance from a node v to t in a SP tree T_e associated with a sidetrack e is denoted by $d_e(v)$.

3 Basic Algorithm

In this section, we propose a simplified way to enumerate the k shortest simple paths. We describe in Section 4 some modifications to achieve our proclaimed running time guarantee.

We initialize an empty priority queue Q that is going to manage candidate paths. The key of a path in Q is its length. We compute the initial SP tree T_0 and push its unique s-t path, represented by an empty sidetrack sequence, to Q. We now process the paths in Q in order of increasing length until we found k simple paths. Let $(e_1, \ldots, e_r)$ be a sidetrack sequence extracted from Q, and p the path that is represented by this sequence. Although the first path that is pushed to Q is always simple, we will eventually push non-simple paths to Q, too. Therefore, we first have to determine whether p is simple in a *pivot step*. This check can be done by simply walking p and marking every visited node.

We first describe how to handle the simple case. We start by outputting p. Let u be the head of e_r, and $T = T_{e_r}$. For every sidetrack $e = (v, w)$ with $v \in \mathrm{suff}(p)$, we discover a new path p' represented by the sequence $(e_1, \ldots, e_r, e)$. We set $\mathrm{dev}(p') = v$, and push p' to Q. By choosing $T_e = T$, we simply reuse the SP tree that is also associated with the last sidetrack in the sequence representing p. The length of p' can easily be computed as $c(p) + \delta_T(e)$. If



(a) Example graph **(b)** Sidetrack sequences

Figure 1 Example for the basic algorithm. In Figure 1a, the thick, solid edges belong to T_0. In Figure 1b, every sidetrack is associated with T_0 except for c', which is associated with the SP tree T_1 comprising the edges b and d. An arrow from sequence p to sequence p' indicates that p is the parent path of p'.

$d_{e_r}(w)$ is undefined because T does not contain a w-t path, we simply ignore e. Apart from these dead ends, we add one path for each sidetrack in $D_T(u)$ to Q. Note that sidetracks emanating from t can safely be ignored.

Consider the example in Figure 1. The sidetrack sequence (a) with $T_a = T_0$ represents a simple path p that passes the nodes s, v_2, v_1, v_3, t in this order. The suffix of this path is its v_2-t subpath, and the sidetracks b, c have tails on this suffix. Therefore, when (a) is extracted from Q, p is output and the sequences (a, b) and (a, c) with $T_a = T_b = T_c$ are pushed to Q.

Now assume we extracted a non-simple path p represented by the sidetrack sequence $(e_1, \ldots, e_r)$. We try to extend the concatenation of $\text{pref}(p)$ and e_r to a simple s-t path. Let $e_r = (v, w)$. Any valid extension avoids the nodes of $\text{pref}(p)$ after v. We are only interested in shortest extensions. Therefore, we compute a new SP tree T and distances d, but in $G - \text{pref}(p)$ instead of G to make sure that nodes of the prefix path of p are not used again. If $w \notin T$, $\text{pref}(p)$ cannot be extended to a simple s-t path, and we discard p. Otherwise, we push the sequence $(e_1, \ldots, e_r)$ to Q again. In this new sequence, we associate T with e_r instead of T_{e_r} from the old sequence. The sequence represents a path p' obtained by concatenating the simple prefix path of p, the edge e_r, and the w-t path in T that, by construction, avoids all nodes of $\text{pref}(p)$. The suffix itself is simple because it is a shortest path in a subgraph of G. Hence, p' is simple. The length of p' is $c(\text{pref}(p)) + c(e_r) + d(w)$.

Consider again the example in Figure 1. The sidetrack sequence (a, c) with $T_a = T_c = T_0$ represents a non-simple path p that visits the nodes s, v_2, v_1, v_3, v_2, v_1, v_3, t in this order. The deviation node of p is v_3, its deviation edge c, and its prefix path is $(a, (v_2, v_1), (v_1, v_3))$. We compute a new SP tree T in $G - \text{pref}(p)$, which only consists of the edge d. Therefore, T does not contain a v_2-t path, and p is discarded. In contrast, assume the sequence (c) with $T_c = T_0$ was just extracted from Q. It represents almost the same path as the sequence above, but it skips the first visit of v_2. Again, v_3 is the deviation node and c the deviation edge. The prefix path comprises the nodes s, v_1 and v_3. After removing them temporarily, a new SP tree T_1 is computed, consisting only of the edges b and d. The sequence (c) with $T_c = T_1$ is pushed to Q. This new sequence represents the simple path $((s, v_1), (v_1, v_3), c, b, d)$, where c is the last sidetrack in the extracted sequence, and (b, d) is the unique v_2-t path in T_1.

Finally, when (c) with $T_c = T_1$ is extracted, the represented path is output. The sidetracks emanating from its prefix are (v_2, v_1) and (v_4, v_3). Since $v_1, v_3 \notin T_1$, these sidetracks are ignored and no new path is pushed to Q.

▶ **Lemma 1.** *The above algorithm computes the k shortest simple s-t paths of a weighted, directed graph $G = (V, E)$.*

Proof. The algorithm uses the same idea of shortest deviations as existing kSSP algorithms or Eppstein's kSP algorithm. We only have to show that a non-simple path p is processed before its simple enhancement p', resulting from the suffix repair in the non-simple case, is actually needed. The set of nodes that are forbidden when the SP tree for p is computed is a proper subset of the node set that the SP tree for p' may not use. The suffix of p is therefore not longer than that of p', and p is extracted from Q (and subsequently, p' is pushed) before we need to extract p'. ◀

This basic form of our algorithm requires too many computations of SP trees:

▶ **Lemma 2.** *The running time of the above algorithm is $\mathcal{O}(km(m + n \log n))$.*

Proof. While processing a non-simple path, at most one new path is pushed to Q, which is always simple. Thus, the parent of a non-simple path is always simple. We have to process at most k simple paths, each of which requires $\mathcal{O}(m + n)$ time. Every simple path may have $\mathcal{O}(m)$ sidetracks extensions. In the worst case, all of them represent non-simple paths, yielding $\mathcal{O}(km)$ SP tree computations with a total running time of $\mathcal{O}(km(m + n \log n))$. The running time for the non-simple cases clearly dominates. For every subset of E, there is at most one permutation of this subset that represents a simple s-t path. The maximum number of paths enumerated by the algorithm is therefore $k' := \min\{k, 2^m\}$. We can limit the size of Q efficiently to k' using a double-ended priority queue [16]. We push $\mathcal{O}(k'm)$ paths to Q and extract $\mathcal{O}(k'm)$ paths from it; both operations require $\mathcal{O}(\log k')$ time on interval heaps. The total time spent on processing Q is $\mathcal{O}(k'm \log k') \subset \mathcal{O}(km^2)$. The pivot step requires $\mathcal{O}(n)$ time for each of the $\mathcal{O}(k'm)$ extracted paths. ◀

Finally, we turn our attention to the space requirements of the above algorithm. We need $\mathcal{O}(n)$ space for each SP tree that we compute. Since SP trees are never discarded and we compute one for each non-simple extracted path, the total space for all SP trees is $\mathcal{O}(kmn)$. For each simple extracted path p, we push a path to Q for each edge that has its tail on p. These new paths are represented by an edge and a pointer to some SP tree, and therefore require constant space. We extract up to k simple paths with $\mathcal{O}(m)$ sidetracks each, and therefore require $\mathcal{O}(km)$ space for Q itself.

4 Improvements

We show how the number of SP tree computations can be reduced to $\mathcal{O}(kn)$ in the worst case. Further, the space requirements are reduced by a factor of n.

So far, we were only able to bound the number of SP tree computations by $\mathcal{O}(m)$ for each extracted simple path. This stems from the fact that there can be $\mathcal{O}(m)$ sidetracks with tails on such a path, each of them requiring a subsequent SP tree computation in the worst case. Consider two sidetrack sequences $(e_1, \ldots, e_r, f_1 = (u, v))$, $(e_1, \ldots, e_r, f_2 = (u, w))$ that were added when a path p represented by $(e_1, \ldots, e_r)$ was processed. Let p_1, p_2 be the paths represented by these sequences, respectively. Assume that both sequences represent non-simple paths, and therefore both require a new SP tree. We assume w.l.o.g. that p_1 is extracted from Q before p_2. When p_1 is extracted from Q, we discover that it contains a cycle. We then have to compute an SP tree T for the graph $G - p'$, where p' is the shortest s-u subpath of p. We push $(e_1, \ldots, e_r, f_1)$ back to Q, and update $T_{f_1} = T$. When p_2 is extracted, the basic algorithm computes an SP tree for the exact same graph. This computation can

therefore be skipped. We check if an SP tree for this graph has already been computed, and reuse it if it exists. In our case, we simply push $(e_1, \dots, e_r, f_2)$ with $T_{f_2} = T$ to Q. We obtain the following result.

▶ **Lemma 3.** *Excluding the time spent on Q, the algorithm proposed in Section 3 in conjunction with SP tree reuse requires $\mathcal{O}(kn(m + n \log n))$ time to process non-simple paths.*

Proof. There are still $\mathcal{O}(km)$ many sequences in Q that represent non-simple paths, but only $\mathcal{O}(kn)$ of them trigger an SP tree computation. Let p be a non-simple path extracted from Q. The initial pivot step requires time $\mathcal{O}(n)$. We store in Q along with each path a pointer to its parent path, as well as a pointer to the SP tree for $G - p'$ for every prefix path p'. We can then check if an SP tree for some prefix path has already been computed, and access it if it has, both in constant time. ◀

The total running time of $\mathcal{O}(km^2)$ spent on Q is no longer dominated. Instead of using a priority queue for the candidate paths, we organize all computed paths in a min-heap in the following way. The shortest path is the root of the min-heap. Whenever a path p' is computed while a path p is processed, we insert p' into the min-heap as a child of p. Figure 1b shows an example of such a min-heap. We use Frederickson's heap selection algorithm [7] to extract the km smallest elements from this heap. The heap described above has maximum degree m, again yielding a running time of $\mathcal{O}(km^2)$. Let P_p be the set of paths found during the processing of p. Instead of inserting every $p' \in P_p$ as a heap child of p, we heapify P_p to obtain the heap H_p, using the lengths of the paths for keys again. The root of H_p is then inserted into the global min-heap as a child of p. Note that the parent path of every path in H_p is not its heap parent in H_p, but still p itself. Every simple path p in the min-heap now has at most two heap successors with the same parent path as p, and at most one heap successor whose parent is p itself. Every non-simple path has at most one simple path as heap processor. The maximum degree of the global min-heap is therefore bounded by three and Frederickson's heap selection can be done in time $\mathcal{O}(km)$.

▶ **Corollary 4.** *The algorithm proposed in Section 3 in conjunction with SP tree reuse and Frederickson's heap selection algorithm computes the k shortest simple s-t paths of a weighted, directed graph $G = (V, E)$, $s, t \in V$, in $\mathcal{O}(kn(m + n \log n))$ time.*

The first improvement above reduced the space required by the basic algorithm from $\mathcal{O}(kmn)$ to $\mathcal{O}(kn^2)$. We are not able to reduce the number of SP tree computations to $o(kn)$. However, it is not necessary to permanently store all these SP trees at the same time. Only up to k of them can contain a simple path that eventually gets extracted from Q. We propose to store the computed SP trees in a max priority queue $\mathcal{S}$. The priority of an SP tree T in $\mathcal{S}$ is $\max\{c(p') + c(e) + d_T(w) \mid e = (v, e) \in E\}$, where $p' = (e_1, \dots, e_r, f)$ is the path T was computed for, and $f = (u, v)$ for some $u \in V$. Whenever $|\mathcal{S}|$ exceeds k, $\mathcal{S}$ contains an SP tree that will not contribute to the k shortest simple paths. This is always the SP tree with the highest priority. It can be extracted from $\mathcal{S}$ in $\mathcal{O}(\log k)$ time and therefore does not have an impact on the worst-case running time. The space that was used to store the extracted tree can later be used to store new SP trees. The number of SP trees stored at any point in time never exceeds $k + 1$. The total space requirements are then dominated by the D_T's, and bounded by $\mathcal{O}(km)$.

Our final improvement does not change the worst-case running time or space consumption. Instead, we will speed up an important part of the algorithm by a factor of n by using a rather cheap test. Consider one of the k simple s-t paths p represented by sidetracks $(e_1, \dots, e_r)$ with $e_i = (v_{i-1}, v_i)$, $s = v_0$ and $t = v_r$. When p is processed, we push the set P_p of paths

to Q, with $|P_p| \in \mathcal{O}(m)$. The basic algorithm tests for each $p' \in P_p$ if p' is simple in time $\mathcal{O}(n)$, leading to a total time of $\mathcal{O}(kmn)$ for these tests. Let $T = T_{e_r}$. By removing all e_i from T, the SP tree decomposes into a set of trees T_i such that T_i is rooted in v_i. The *block* i is the node set of T_i. Observe that the path p' represented by a sequence $(e_1, \ldots, e_r, e)$, $e = (v_i, w)$, with v_i, w in block i, j, respectively, is simple iff $i < j$. If $i \geq j$, we follow p until we reach v_i, traverse e and follow T to reach v_i again. Otherwise, the first node on p we hit after deviating from it via e is v_j. Since $i < j$, the v_j-t subpath of p does not contain v_i, so p' is simple. The partition of V into blocks is $\mathcal{O}(n)$-time computable. We can then collect all sidetracks deviating from p and check for each of them if their heads belong to a smaller block than their tails in $\mathcal{O}(m)$ total time. We store this information along with the corresponding sidetrack sequences in Q. The pivot turn is replaced by a constant time lookup. All tests for simplicity then require time $\mathcal{O}(k(m+n))$ instead of $\mathcal{O}(kmn)$.

5 Experiments

To demonstrate the effectiveness of our algorithm, we conducted a series of experiments. Feng [6] showed recently that their algorithm is the most efficient one in practice. We therefore compare our sidetrack-based algorithm to Feng's node classification algorithm. For reference, we also include results for the most promising third contender, an algorithm proposed by Sedeño-Noda [17]. We used all graph classes that Feng had used in their experiments, including road graphs that are especially relevant in practice.

Sedeño-Noda kindly provided us with the implementation KCM of their algorithm. We conducted our experiments on a desktop computer very similar to that of Feng. On our computer, KCM was consistently slower than what is reported for KCM on Feng's computer [6]. On average, we required 10.4 seconds on NY and 15.94 seconds on BAY (described in Section 5.2) using KCM; Feng reported 8.81 and 11.23 seconds, respectively. In contrast, our implementation NC of Feng's algorithm (*without* express edges) consistently gives lower running times than those reported by Feng. We also implemented Feng's algorithm *with* express edges, which was always slower than NC. Note that Feng did not specify whether express edges were used in their experiments. All improvements proposed in Section 4 were used in the implementation SB of our sidetrack-based algorithm with the following exception: Frederickson's heap selection algorithm was used neither for NC nor for SB. This results in an additional running time of $\mathcal{O}(km \log k)$ for SB, but not for NC.

Shortest paths (NC) and SP trees (SB) are computed using a common implementation of Dijkstra's algorithm; tentative labels are managed by a pairing heap. Our implementation of Dijkstra's algorithm stops as soon as the label of the target node is made permanent if only a single pair shortest path is needed, which is essential for NC. SP trees are computed lazily. A tree is initialized without any edges, and the source node is pushed to a priority queue that is permanently associated to the tree. Whenever a part of the tree is queried (i.e. the distance or predecessor of some node) that has not yet been computed, we simulate the Dijkstra algorithm using the associated priority queue until the queried part is settled. The queue of candidate paths Q is implemented as an interval heap, a form of double-headed priority queues, which allows us to limit its size efficiently to the number of simple paths that have yet to be output. For SB, we use separate priority queues Q_s and Q_n for simple and non-simple paths, respectively. Whenever a path has to be extracted, SB extracts from Q_n iff the shortest path in Q_n is cheaper than the shortest path in Q_s.

We implemented NC and SB in C++, using forward and reverse star representation for directed graphs. The experiments ran on an Intel Core i7-3770 @ 3.40GHz with 16GB of

RAM on a GNU/Gentoo Linux with kernel version `4.4.6` and TurboBoost turned off. Source code was compiled using the GNU C++ compiler `g++-4.9.3` and `-O3` optimization.

5.1 Random Graphs

We first considered random graphs generated by the `sprand` generator provided on the website of the Ninth DIMACS Implementation Challenge [3]. The generator draws at random a fixed amount of edges, possibly resulting in a multigraph. For each combination of graph size $n \in \{2000, 4000, 6000, 8000, 10\,000\}$ and *linear density* $m/n \in \{2, 3, 4, 7, 10, 20, 30, 40, 50\}$, we generated 20 random graphs, and enumerated $k \in \{200, 500, 1000, 2000\}$ simple paths. Edge weights were selected uniformly from $\{1, \dots, 10\,000\}$. In Table 1, the median and 90% quantile $Q_{.9}$ of execution times for some densities and $k = 2000$ are summarized. Our results confirm Feng's claim that NC is usually faster than KCM on random graphs. NC seems to struggle with very low densities of $m = 2n$ and gets faster for graphs with densities up to about $m = 30n$. On the other hand, KCM and SB display a more consistent growth. SB is always the fastest of the three, with speedup factors ranging from 8 to 15 for lower densities, and 7.5 to 25 for higher densities when compared to the second-fastest algorithm.

We now consider the dispersion of the three algorithms. For SB, 90% of the instances finish within 160% of the corresponding median running times (the fastest 50%) for most combinations of n and m. For KCM, this ratio stays below 106% for all but two combinations of n and m. We could not find any correlation between n, m and k on the one side, and the dispersion of running times on the other side, for KCM and SB. In contrast, NC regularly requires more than thrice the median running time to answer 90% of the queries. Running times are therefore much harder to predict when using NC instead of SB or KCM.

Table 2 shows the median number of Dijkstra calls. The numbers are relatively stable across the various densities, but the Dijkstra counts for the SB algorithms is orders of magnitudes smaller than the count for the NC algorithm. Note, however, that SB needs to compute the complete SP tree every time. In contrast, NC only solves single pair shortest path problems on rather small subgraphs. We also provide the number of polls, i.e., the total number of nodes that were extracted from Dijkstra's priority queue, for comparability. The ratio of the number of polls of NC and SB ranges from 4.6 to 50, and suggests that saving SP tree computations is much more beneficial than reducing the number of nodes visited to answer single-pair shortest path queries.

Finally, the number of SP tree computations actually declines as n grows. Recall that, in the worst case, we have to compute one SP tree for each node of each output simple path. Table 2 shows results for $k = 2000$ and $n \geq 2000$. Nevertheless, the median number of SP tree computations does not exceed 65. Most simple paths therefore correspond to those well-behaved cases where paths represented by sidetracks in already computed SP tree areas are themselves simple most of the time.

5.2 Road Graphs

We consider road graphs of various areas in the USA called TIGER graphs, again provided by the DIMACS website [3]. In particular, we use the road networks of New York (NY), the San Francisco Bay Area (BAY), Colorado (COL), and Florida (FLA). For each of the four areas, we drew 20 s-t pairs at random and enumerated $k \in \{100, 200, 300\}$ paths. The resulting running times are summarized in Table 3, along with the median number of polls. With respect to the median running times, KCM is clearly dominated by NC on any input class, which in turn is dominated by SB. SB achieves a minimum speedup of around 8 on

Table 1 Median and 90% quantile $Q_{.9}$ of running times in seconds for random graphs, $k = 2000$.

		$m = 2n$		$m = 4n$		$m = 10n$		$m = 30n$		$m = 50n$	
n		Med.	$Q_{.9}$	Med.	$Q_{.9}$	Med.	$Q_{.9}$	Med.	$Q_{.9}$	Med.	$Q_{.9}$
2000	KCM	0.67	0.69	0.85	0.88	1.29	1.32	3.58	4.04	9.77	14.11
	NC	0.99	2.51	0.48	1.25	0.39	1.25	0.47	1.50	2.15	3.84
	SB	**0.09**	**0.13**	**0.08**	**0.10**	**0.10**	**0.15**	**0.17**	**0.20**	**0.23**	**0.31**
4000	KCM	1.39	1.46	1.79	1.87	2.96	3.06	15.64	16.03	32.88	33.17
	NC	1.01	2.88	0.84	2.71	0.82	1.97	1.33	5.06	2.07	5.49
	SB	**0.11**	**0.12**	**0.09**	**0.13**	**0.12**	**0.19**	**0.19**	**0.22**	**0.26**	**0.36**
6000	KCM	2.19	2.25	2.86	2.90	5.50	5.78	30.13	30.61	53.33	54.51
	NC	3.28	6.44	0.53	1.67	0.71	3.36	2.05	7.92	2.22	9.07
	SB	**0.13**	**0.20**	**0.11**	**0.13**	**0.13**	**0.21**	**0.22**	**0.32**	**0.30**	**0.45**
8000	KCM	3.04	3.06	4.08	4.14	12.37	12.60	43.36	45.31	73.11	75.00
	NC	1.79	7.84	0.68	2.66	1.92	4.60	3.49	9.67	2.55	9.66
	SB	**0.12**	**0.28**	**0.10**	**0.14**	**0.16**	**0.21**	**0.24**	**0.34**	**0.32**	**0.38**
10 000	KCM	3.96	3.98	5.48	5.53	15.71	15.84	55.95	57.76	92.81	94.79
	NC	1.86	11.91	1.17	5.44	2.61	9.02	6.31	13.56	9.68	26.23
	SB	**0.13**	**0.18**	**0.14**	**0.24**	**0.17**	**0.21**	**0.25**	**0.30**	**0.36**	**0.48**

Table 2 Median number of Dijkstra calls and polls (in millions) of NC and SB for random graphs, $k = 2000$.

		$m = 4n$		$m = 10n$		$m = 30n$		$m = 50n$	
n		Dijkstras	Polls	Dijkstras	Polls	Dijkstras	Polls	Dijkstras	Polls
2000	NC	16 272	1.08	14 533	0.60	13 939	0.43	14 510	2.09
	SB	46	**0.09**	65	**0.13**	38	**0.08**	44	**0.09**
4000	NC	17 292	1.93	14 581	1.45	15 805	1.20	15 605	1.39
	SB	25	**0.10**	20	**0.08**	21	**0.08**	29	**0.11**
6000	NC	17 499	0.86	16 652	0.70	16 544	1.52	16 444	1.13
	SB	23	**0.14**	19	**0.11**	24	**0.14**	21	**0.13**
8000	NC	18 300	1.01	17 316	2.40	17 127	2.72	17 034	1.09
	SB	16	**0.12**	17	**0.13**	17	**0.14**	17	**0.14**
10 000	NC	19 074	1.94	17 824	3.53	18 125	5.07	18 187	6.11
	SB	16	**0.15**	15	**0.15**	10	**0.10**	14	**0.13**

Table 3 Median and 90% quantile $Q_{.9}$ of running times in seconds, and median number of polls for large TIGER road graphs. KCM does not provide the number of polls and was not able to compute the 300 shortest simple paths on FLA.

		$k = 100$			$k = 200$			$k = 300$		
Area		Med.	$Q_{.9}$	Polls	Med.	$Q_{.9}$	Polls	Med.	$Q_{.9}$	Polls
NY	KCM	9.72	11.57	-	19.54	23.48	-	29.76	35.55	-
	NC	2.09	12.38	3.90	3.80	24.30	6.91	5.43	36.18	9.76
	SB	**0.18**	**1.57**	**0.53**	**0.26**	**3.92**	**0.69**	**0.43**	**5.99**	**1.06**
BAY	KCM	12.72	25.90	-	25.16	53.00	-	38.17	83.36	-
	NC	5.32	17.88	15.21	9.49	34.57	28.77	14.14	51.11	38.72
	SB	**0.30**	**4.28**	**0.96**	**0.55**	**9.67**	**1.58**	**0.71**	**14.17**	**1.90**
COL	KCM	17.13	32.99	-	34.56	71.66	-	56.77	117.12	-
	NC	6.83	27.71	16.65	12.04	49.23	30.30	16.73	65.92	44.23
	SB	**0.17**	**11.80**	**0.44**	**0.22**	**17.43**	**0.44**	**0.29**	**26.64**	**0.44**
FLA	KCM	48.51	99.06	-	95.69	215.48	-	-	-	-
	NC	29.86	70.10	54.21	58.07	132.73	106.02	85.70	193.30	157.40
	SB	**0.47**	**4.31**	**1.07**	**0.58**	**20.84**	**1.07**	**0.71**	**26.99**	**1.07**

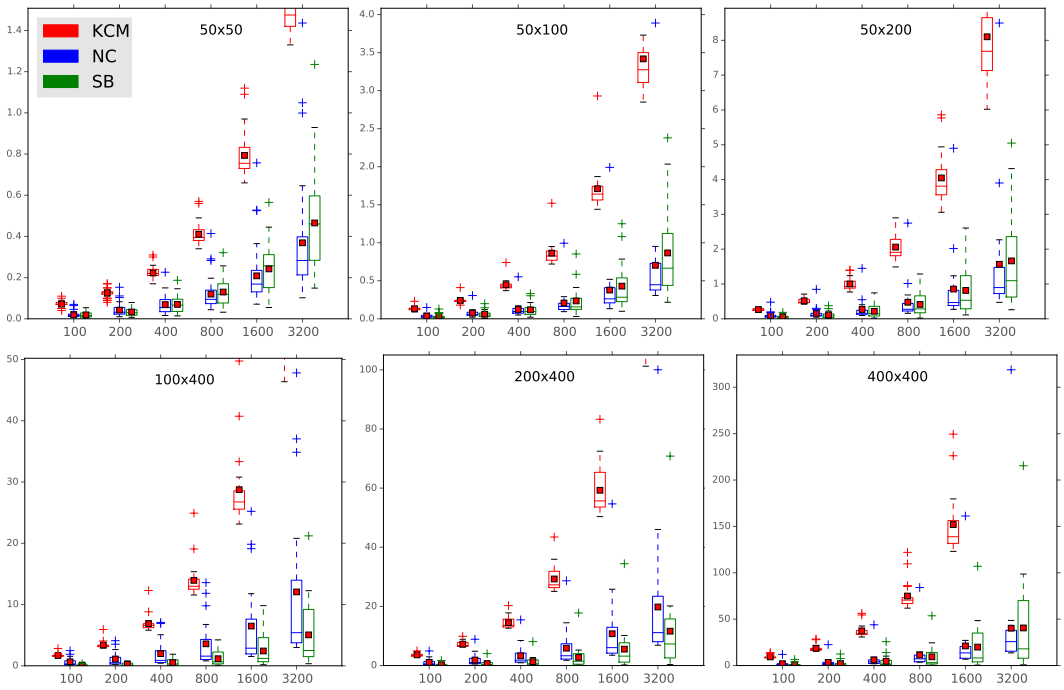


Figure 2 Boxplots of running times in seconds for grid graphs. Plus signs represent outliers. A red square marks the mean. The x-axis corresponds to different values of k and uses a logarithmic scale.

NY for $k = 100$ in comparison to NC. In 90% of the input classes, however, SB achieves a speedup factor of more than 62, and even peaks at a speedup factor of 120. Running times of KCM and SB are much more dispersed than in the random graph case. Still, SB answers 90% of the queries in about the same time the 50% fastest query times of NC, and sometimes much faster. KCM was not able to finish all computations on all inputs.

5.3 Grid Graphs

We repeated Feng's experiments on grid graphs generated by the `spgrid` generator provided on the DIMACS website [3]. The grids have sidelengths $l, w \in \{50, 100, 200, 400\}$ with $l \leq w$, resulting in 10 different grids. For each grid, we generated 20 weight functions by selecting uniformly from $\{1, \ldots, 10\,000\}$ for each edge, and then enumerated $k \in \{100, 200, 400, 800, 1600, 3200\}$ paths. The results of our experiments on grid graphs are summarized in Figure 2. KCM is again the slowest algorithm, but NC is not slower than SB any more. Although NC and SB differ on some classes, there does not seem to be any correlation to the shape or size of the grid. For example, NC is slightly faster on 50×100 grids, but slower on 200×400 grids although these two configurations share the same shape, characterized by an aspect ratio of 2. On the other hand, NC loses its advantage over SB when the grid grows from 50×50 to 50×200 (upper row of plots), but SB loses its advantage over NC as it grows from 100×400 to 400×400 (lower row). In summary, none of the two algorithms is clearly better than the other on grid graphs.

The algorithm proposed in this paper is not slower on any of the considered graph classes, and even faster than state-of-the-art algorithms on random and TIGER road graphs by an order of magnitude. Our algorithm shines on graphs where the length of shortest paths is small in relation to the graph size because sidetrack heaps tend to be smaller.

References

1 Aaron Bernstein. A nearly optimal algorithm for approximating replacement paths and k shortest simple paths in general graphs. In *SODA 2010*, pages 742–755. SIAM, 2010.

2 S. Clarke, A. Krikorian, and J. Rausen. Computing the N best loopless paths in a network. *J. SIAM*, 11(4):1096–1102, 1963.

3 The Ninth DIMACS Implementation Challenge: 2005-2006. `http://www.dis.uniroma1.it/challenge9/`. Accessed: 2015-11-12.

4 David Eppstein. Finding the k shortest paths. *SIAM J. Comput.*, 28(2):652–673, 1998.

5 David Eppstein. K-best enumeration. arXiv: 1412.5075v1 [cs.DS], 2014.

6 Gang Feng. Finding k shortest simple paths in directed graphs: A node classification algorithm. *Networks*, 64(1):6–17, 2014.

7 Greg N. Frederickson. An optimal algorithm for selection in a min-heap. *Inf. Comput.*, 104(2):197–214, 1993.

8 Asaf Frieder and Liam Roditty. An experimental study on approximating k shortest simple paths. *ACM J. Exp. Algorithmics*, 19(1), 2014.

9 Zvi Gotthilf and Moshe Lewenstein. Improved algorithms for the k simple shortest paths and the replacement paths problems. *Inf. Process. Lett.*, 109(7):352–355, 2009.

10 John Hershberger, Matthew Maxel, and Subhash Suri. Finding the k shortest simple paths: A new algorithm and its implementation. *ACM Trans. Algorithms*, 3(4), 2007.

11 Naoki Katoh, Toshihide Ibaraki, and H. Mine. An efficient algorithm for K shortest simple paths. *Networks*, 12(4):411–427, 1982.

12 Marta M. B. Pascoal. Implementations and empirical comparison of K shortest loopless path algorithms, 2006. 9th DIMACS Implementation Challenge Workshop: Shortest Paths.

13 Seth Pettie. A new approach to all-pairs shortest paths on real-weighted graphs. *Theor. Comput. Sci.*, 312(1):47–74, 2004.

14 Liam Roditty. On the k shortest simple paths problem in weighted directed graphs. *SIAM J. Comput.*, 39(6):2363–2376, 2010.

15 Liam Roditty and Uri Zwick. Replacement paths and k simple shortest paths in unweighted directed graphs. *ACM Trans. Algorithms*, 8(4):33, 2012.

16 Sartaj Sahni. *Data Structures, Algorithms, and Applications in C++*. McGraw-Hill Pub. Co., 1st edition, 1999.

17 Antonio Sedeño-Noda. An efficient time and space K point-to-point shortest simple paths algorithm. *Appl. Math. and Comput.*, 218(20):10244–10257, 2012.

18 Virginia Vassilevska Williams and Ryan Williams. Subcubic equivalences between path, matrix and triangle problems. In *FOCS 2010*, pages 645–654, 2010.

19 Jin Y. Yen. Finding the k shortest loopless paths in a network. *Networks*, 17(11):712–716, 1971.