A Sidetrack-Based Algorithm For Finding the \( k \) Shortest Simple Paths in a Directed Graph

Denis Kurz, Petra Mutzel

Department of Computer Science, TU Dortmund, Germany
{denis.kurz,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 [14] 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 \( O(m) \) simple paths with a single shortest path tree computation and \( O(n + m) \) 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. Further, the running time is far better predictable due to very small dispersion.

Keywords: directed graph, \( k \)-best, shortest path, simple path, weighted graph

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 \( t \) 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 \( O(m + n \log n + k \log k) \) and is due to Eppstein [3].

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 \( O(m + n \log n) \). This can even be reduced to \( O(m + n) \) if the shortest path tree (SP tree) is given in the input or if the SP tree can be computed in time \( O(m+n) \). 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 \( O(k \log k) \) 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 \cite{6} can be used to enumerate the paths after the initialization phase in time $O(k)$.

The $k$ shortest simple path problem ($k$SSP), introduced by Clarke, Krikorian and Schwartz \cite{2}, seems to be more expensive, computationally. In contrast to $k$SP, the computed 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 \cite{14} used to have the best theoretical worst-case running time of $O(kn(m + n \log n))$ for quite some time. Gotthilf and Lewenstein \cite{7} improved upon this bound recently. They observed that $k$SSP can be solved by solving $O(k)$ all pairs shortest path (APSP) instances. Using the APSP algorithm by Pettie \cite{11}, they obtain a new upper bound of $O(kn(m + n \log n \log \log n))$. Vassilevska Williams and Williams \cite{13} showed that, for constant $k$, an algorithm for $k$SSP with running time $O(n^3 \varepsilon^{-\varepsilon})$ for some positive $\varepsilon$ (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 \cite{4}. The $k$SSP on undirected graphs seems to be significantly easier. Katoh et al. \cite{9} proposed an algorithm that solves $k$SSP on undirected graphs in time $O(k(m + n \log n))$.

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 $i$th. This problem has to be solved $O(k)$ 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 $O(|p|)$ shortest path computations, resulting in time $O(n(m + n \log n))$. Hershberger et al. \cite{8} 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 $O(m + n \log n)$ 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 $O(n(m + n \log n))$.

Some approaches reuse one fixed reversed SP tree $T_0$ rooted in $t$ and computed during the initialization of their $k$SSP algorithm, in contrast to $O(1)$ SP trees per replacement path instance. Pascoal \cite{10} 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 $O(m + n)$ per replacement path instance by partitioning the nodes into blocks as described by Hershberger et al. \cite{8}. Feng \cite{5} 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 \( O(n(m + n \log n)) \) for each replacement path instance.

Our contribution. We propose an algorithm that was derived from Eppstein’s notion of a path graph \([3]\). 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 simple paths from one shortest path tree computation. If the underlying graph is acyclic, the revised algorithm at the end of this paper requires \( O(n \log n + k(m + n)) \) 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. Our algorithm works on multigraphs without modification. This is also true for every other \( k\)SSP algorithm we know of. After some definitions in Section 2, we propose a simplified version of our algorithm with running time \( O(km(m + n \log n)) \) in Section 3. In Section 4, we show how this running time can be reduced to \( O(km(m + n \log n)) \), and how to reduce the number of shortest path tree computations in practice. 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 \( c : E \to \mathbb{R}_+^3 \) 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 \( p = (e_1, e_2, \ldots, e_r) \), \( e_i = (v_i, w_i) \), with \( v = v_1 \), \( w = w_r \) and \( w_i = v_{i+1} \) for \( 1 \leq i < r \). 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 \( 1 \leq i, j \leq r \), \( p \) is a simple path. The prefix \( (e_1, \ldots, e_i) \) 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 \( G \setminus p \) to denote the induced subgraph \( G \{ \{ v \in V \mid v \notin p \} \} \).

The \( k \) shortest simple path problem (\( k\)SSP) 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 \( c(p) \leq c(p') \) for every pair \( p \in P, p' \notin P \) of simple paths. We obtain the \( k \) shortest path problem (\( k\)SP) 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 $v$-$t$ path in $G$ exists. We denote the latter case by $v \not\in T$ even if $v$ is in the node set of $T$. Our algorithm will compute several SP trees, the first of which we call initial SP tree $T_0$. An edge $e \not\in T$ 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 dev($p'$), and $e_{i^*}$ is the deviation edge of $p'$. As is quite usual for kSSP algorithms, we will discover paths in a hierarchical fashion: a path $p'$ is added to the candidate set after $p$ was extracted from the candidate set. In such cases, $p$ is called 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 $\text{pref}(p) := p'^*$ starting in $s$, and its suffix path $\text{suffix}(p)$ 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 introduce a generalization of Eppstein’s representation [3] for paths. Eppstein represented paths as sequences of sidetracks, which were all sidetracks w.r.t. the same shortest path tree. In our representation, every sidetrack $e$ in a sidetrack sequence may be associated with a different shortest path 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 $e_1$, we traverse $e_1$ and follow $T_{e_1}$ until we reach the tail of $e_{i^*}$, 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 shortest path tree $T_e$ associated with a sidetrack $e$ is denoted by $d_e(v)$.

3 Basic Algorithm

In this section, we propose a rather simple way to enumerate the $k$ shortest simple paths. We will describe later how to modify this algorithm 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 shortest path 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 \(p\). For every sidetrack \(e = (u, v)\) with \(u \in \text{suff}(p)\), we discover a new path \(p'\) represented by the sequence \((e_1, \ldots, e_r, e)\). We set \(\text{dev}(p') = u\), and push \(p'\) to \(Q\). The length of \(p'\) can easily be computed as \(c(p) + \delta_{T_e}(e)\). If \(d_{e_r}(v)\) is undefined because \(T_e\) does not contain a \(v\)-\(t\) path, we simply ignore \(e_r\). By choosing \(T_e = T_{e_r}\), we simply reuse the shortest path tree that is also associated with the last sidetrack in the sequence representing \(p\). Note that sidetracks emanating from \(t\) can safely be ignored.

Consider the example in Fig. 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 \(v_2\)-\(t\) sub-path, and the sidetracks \(b\) and \(c\) 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 the prefix path of \(p\) and \(e_r\) to a simple \(s\)-\(t\) path. Let \(e_r = (v, w)\). Any valid extension has to avoid the nodes of \(\text{pref}(p)\) after \(v\), and 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 simply discard \(p\). Otherwise, we push the sequence \((e_1, \ldots, e_r)\) to \(Q\) again. In this new sequence, however, 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, too, because it is a shortest path in a subgraph of $G$. Hence, $p'$ is simple. The length of this path is $c(\text{pref}(p)) + c(e_r) + d(w)$.

Consider again the example in Figure 1. The sidetrack sequence $(a, c)$ with $T_0 = T'_0 = T_1$ 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 $T'_0 = T_0$ 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 shortest path tree $T_1$ is computed, consisting only of the edges $b$ and $d$. The sequence $(c)$ with $T_0 = T_1$ is pushed to $Q$. This new sequence represents the simple path $((s, v_1), (v_1, v_3), c, b, d)$, i.e., the concatenation of the prefix path of $p$, the last sidetrack $c$ in the extracted sequence, and the unique $v_2$-$t$ path in $T_1$.

Finally, when $(c)$ with $T'_0 = T_1$ 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 $k$SSP algorithms or Epstein’s $k$SP 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 $p'$.

In terms of running time, the above algorithm requires too many computations of SP trees.

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

**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 $O(m + n)$ running time. Every simple path may have $O(m)$ sidetrack extensions. In the worst case, all of them represent non-simple paths, yielding $O(km)$ SP tree computations with a total running time of $O(km(m + n \log n))$ if Dijkstra’s algorithm with Fibonacci heaps is used. 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 \([12]\). We push \( O(k'm) \) paths to \( Q \) and extract \( O(k'm) \) paths from it; both operations require \( O(\log k') \) time on interval heaps. The total time spent on processing \( Q \) is then \( O(k'm \log k') \subset O(km^2) \).

The pivot step requires \( O(n) \) running time for each of the \( O(k'm) \) extracted paths.

\[ \square \]

4 Improvements

We show how the number of SP tree computations can be reduced to \( O(kn) \) in the worst case, and seemingly even further in practice.

So far, we were only able to bound the number of SP tree computations by \( O(m) \) for each extracted simple path. This stems from the fact that there may be \( O(m) \) sidetracks from 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 \( s-u \) sub-path of \( p \). We push \((e_1, \ldots, e_r, f_1)\) back to \( Q \), updating \( T_{f_1} = T \). When \( p_2 \) is extracted, the basic algorithm computes an SP tree for the exact same graph. This computation may 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 \((e_1, \ldots, e_r, f_2)\) 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 \( O(kn(m + n \log n)) \) time to process non-simple paths.

Proof. There are still \( O(km) \) many sequences in \( Q \) that represent non-simple paths, but only \( O(kn) \) of them trigger an SP tree computation. Let \( p \) be a non-simple path extracted from \( Q \). The initial pivot step requires time \( O(n) \). If each path in \( Q \) manages a pointer to its parent path as well as a pointer to the SP tree for \( G - p' \) for every prefix path \( p' \), already computed SP trees can be accessed in constant time.

\[ \square \]

The total running time of \( O(km^2) \) spent on \( Q \) is now 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 1(b) shows an example of such a min-heap.
We want to extract the \( km \) smallest elements from this heap using Frederickson's heap selection algorithm \[4\]. 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 1.** 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.

We propose two more modifications that do not change the asymptotic worst-case running time. We provide evidence that these changes make the algorithm faster in practice.

Consider one of the \( k \) simple \( s-t \) paths \( p \) represented by sidetracks \((e_1, \ldots, e_r)\) 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 \( v_j \) 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 \( v_i \), and therefore, \( p' \) is simple. The partition of \( V \) into blocks can be computed in time \( \mathcal{O}(n) \). 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 \( \mathcal{O}(kmn) \).

Finally, we want to reduce the number of SP tree computations in practice. Let \( p \) be a non-simple path represented by the sequence \((e_1, \ldots, e_r)\) with \( e_i = (v_{i-1}, v_i) \). After we discover that \( p \) is not simple, the basic algorithm computes an SP tree in \( G - \text{pref}(p) \). Only then does the algorithm check if \( v_r \in T \).

Obviously, there is a shortest \( w-t \) path in \( G - \text{pref}(p) \), i.e., \( w \in T \), iff there is some directed path from \( w \) to \( t \) in \( G - \text{pref}(p) \). Latter can be checked by a much simpler reachability check in time \( \mathcal{O}(m+n) \). A naive approach checks reachability for every combination of some node \( v \in V \) and one of the \( \mathcal{O}(kn) \)
Table 1: Median and 90% quantile $Q_{.9}$ of running times in seconds on random graphs with $k = 2000$.

| n     | $m = 2n$ | $m = 4n$ | $m = 10n$ | $m = 30n$ | $m = 50n$ |
|-------|----------|----------|-----------|-----------|-----------|
|       | Med   | $Q_{.9}$ | Med   | $Q_{.9}$ | Med   | $Q_{.9}$ | Med   | $Q_{.9}$ | Med   | $Q_{.9}$ |
| 2,000 | NC     | 0.91    | 2.24    | 0.41    | 1.96    | 0.35    | 1.14    | 0.41    | 1.39    | 1.95    | 9.41    |
|       | SB-r   | 0.18    | 0.23    | 0.27    | 0.29    | 0.72    | 0.75    | 3.70    | 3.86    | 8.94    | 9.08    |
|       | SB-o   | 0.09    | 0.17    | 0.07    | 0.09    | 0.09    | 0.12    | 0.11    | 0.16    | 0.16    | 0.29    |
| 4,000 | NC     | 0.90    | 2.63    | 0.76    | 2.39    | 0.75    | 1.79    | 1.24    | 4.61    | 1.92    | 4.92    |
|       | SB-r   | 0.35    | 0.40    | 0.53    | 0.58    | 1.39    | 1.60    | 7.26    | 7.31    | 17.46   | 17.64   |
|       | SB-o   | 0.16    | 0.21    | 0.12    | 0.17    | 0.13    | 0.20    | 0.17    | 0.22    | 0.27    | 0.38    |
| 6,000 | NC     | 2.99    | 0.88    | 0.47    | 1.94    | 0.65    | 3.06    | 1.98    | 7.14    | 2.07    | 8.34    |
|       | SB-r   | 0.54    | 0.61    | 0.81    | 0.83    | 2.11    | 2.18    | 10.95   | 11.08   | 26.53   | 26.70   |
|       | SB-o   | 0.23    | 0.30    | 0.21    | 0.26    | 0.20    | 0.27    | 0.27    | 0.37    | 0.31    | 0.50    |
| 8,000 | NC     | 1.62    | 7.16    | 0.62    | 2.52    | 1.79    | 4.27    | 3.29    | 9.23    | 2.45    | 8.94    |
|       | SB-r   | 0.69    | 0.86    | 1.08    | 1.11    | 2.81    | 2.89    | 14.66   | 15.35   | 34.84   | 35.50   |
|       | SB-o   | 0.28    | 0.46    | 0.23    | 0.26    | 0.32    | 0.41    | 0.34    | 0.54    | 0.39    | 0.47    |
| 10,000| NC     | 1.70    | 10.86   | 1.09    | 5.00    | 2.46    | 8.97    | 5.83    | 12.19   | 8.96    | 23.97   |
|       | SB-r   | 0.87    | 0.92    | 1.37    | 1.48    | 3.63    | 3.81    | 18.42   | 18.64   | 43.59   | 43.92   |
|       | SB-o   | 0.35    | 0.40    | 0.30    | 0.42    | 0.33    | 0.38    | 0.37    | 0.44    | 0.45    | 0.61    |

5 Experiments

To demonstrate the effectiveness of our algorithm, we conducted a series of experiments. Feng [5] showed recently that their algorithm is the most efficient one in practice. We therefore only compare our algorithm to Feng’s node classification algorithm (NC). Our implementation of NC does not use express edges and achieves better running times than the implementation of Feng, who appears to have used the same processor as we did. We implemented two variants of
Table 2: Median number of Dijkstra calls and polls for random graphs.

| k   | m   | n   | Dijkstra | Polls | Dijkstra | Polls | Dijkstra | Polls |
|-----|-----|-----|----------|-------|----------|-------|----------|-------|
| 2000| 2m  | 2   | 22,984   | 4402,180 | 14,532 | 615,883 | 14,512   | 2,150,588 |
|     |     | 4   | 80       | 158,740  | 65     | 129,426 | 44       | 87,640   |
|     |     | 8   | 82       | 162,774  | 65     | 129,426 | 44       | 87,640   |
| 4000| 10m  | 2m  | 25,130   | 3,284,822 | 14,589 | 1,465,676 | 15,004 | 1,404,670 |
|     |     | 4   | 44       | 177,236  | 20     | 77,830  | 28       | 113,790  |
|     |     | 8   | 46       | 185,205  | 20     | 77,830  | 28       | 113,790  |
| 6000| 20m  | 2m  | 26,990   | 12,342,440 | 16,652 | 719,438 | 16,444 | 1,151,669 |
|     |     | 4   | 44       | 263,158  | 18     | 110,849 | 21       | 125,826  |
|     |     | 8   | 44       | 263,158  | 18     | 110,849 | 21       | 125,826  |
| 8000| 40m  | 2m  | 26,810   | 7,478,962 | 17,416 | 2,418,042 | 17,034 | 1,199,572 |
|     |     | 4   | 25       | 199,528  | 16     | 131,850 | 17       | 135,853  |
|     |     | 8   | 25       | 199,528  | 16     | 131,850 | 17       | 135,853  |
| 10000| 50m  | 2m  | 26,633   | 4,806,279 | 17,825 | 3,552,732 | 18,186 | 6,130,731 |
|     |     | 4   | 23       | 229,591  | 15     | 149,870 | 14       | 134,892  |
|     |     | 8   | 23       | 229,634  | 15     | 149,870 | 14       | 134,892  |

5.1 Random Graphs

We first considered random graphs generated by the `sprand` generator provided on the website of the ninth DIMACS implementation challenge [1]. 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.
In Table 1, the median and 90% quantile $Q_{0.9}$ of execution times for some densities are summarized. For small densities, we observe that SB-r is faster than NC, but becomes much slower as the density grows. The running time of SB-r increases by a factor of 50 between $m = 2n$ and $m = 50n$, but only by a factor of 2 for NC. SB-o is about twice as fast as SB-r for very small densities, and is even more robust against density changes than NC. SB-o is thus the fastest of the three algorithms for all graph sizes and densities. Also note the very low dispersion of SB running times. For NC, the 90% quantile of the running time is regularly three times the median running time, and even exceeds a factor of 6 for $n = 10,000$ and $m = 20,000$. In contrast, this quotient is always much closer to 1 for SB-o and assumes its maximum of 1.87 for $n = 2000$, $m = 100,000$. We can therefore predict the running time of SB-o much more accurate than that of NC. The $Q_{0.9}$ running time of SB-o is still well below the median running time of both SB-r and NC. The dispersion of SB-r is even lower.

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. NC still requires an order of magnitude more polls compared to SB. Further, there are only small differences in the median number of polls for SB-r and SB-o for $m = 2n$. Solving incremental reach for each of the 2000 output simple paths only reduced the number of SP tree computations by at most three for $n = 8000$, $m = 2n$. The
extra effort involved to reduce SP tree computations ranges from 50% of the
total running time on very sparse graphs to 98% on very dense graphs, and does
not pay off. Therefore, SB-o is clearly the fastest algorithm on random graphs.

5.2 Road Graphs

We considered road graphs of various areas in the USA called TIGER graphs,
again provided by the DIMACS website \[1\]. In particular, we used the road
networks of New York (NY), the San Francisco Bay Area (BAY), Colorado
(COL), and Florida (FLA). The sizes of these graphs are shown in [Table 3(a)]

We drew 20 s-t pairs at random and enumerated \( k \in \{100, 200, 300\} \) paths.
The resulting running times are summarized in [Table 4(a)] along with the
median number of polls. The median running time of NC is clearly dominated by
both SB variants. SB-o achieves a minimum speedup around 4 on NY across all
values of \( k \); on FLA, the speedup is roughly 12. SB-r takes approximately twice
the time of SB-o and is still much faster than NC. The ratio of \( Q_{90} \) and median
running time is worse for SB, but the \( Q_{90} \) time itself of SB-o is still better than
that of NC except for 300 paths on COL. On FLA, the largest graph, the 90% quantile of SB-o is much better than the median running time of NC.

References

1. The ninth dimacs implementation challenge: 2005-2006. \[ http://www.dis.uniroma1.it/challenge9/ \] accessed: 2015-11-12
2. Clarke, S., Krikorian, A., Rausen, J.: Computing the N best loopless paths in a
network. J. SIAM 11(4), 1096–1102 (1963)
3. Eppstein, D.: Finding the k shortest paths. SIAM J. Comput. 28(2), 652–673 (1998)
4. Eppstein, D.: K-best enumeration (2014), arXiv: 1412.5075v1 [cs.DS]
5. Feng, G.: Finding k shortest simple paths in directed graphs: A node classification
algorithm. Networks 64(1), 6–17 (2014)
6. Frederickson, G.N.: An optimal algorithm for selection in a min-heap. Inf. Comput.
104(2), 197–214 (1993)
7. Gotthilf, Z., Lewenstein, M.: Improved algorithms for the k simple shortest paths
and the replacement paths problems. Inf. Process. Lett. 109(7), 352–355 (2009)
8. Hershberger, J., Maxel, M., Suri, S.: Finding the k shortest simple paths: A new
algorithm and its implementation. ACM Transactions on Algorithms 3(4) (2007)
9. Katoh, N., Ibaraki, T., Mine, H.: An efficient algorithm for K shortest simple paths.
Networks 12(4), 411–427 (1982)
10. Pascoal, M.M.B.: Implementations and empirical comparison of K shortest loop-
less path algorithms (2006), 9th DIMACS Implementation Challenge Workshop:
Shortest Paths
11. Pettie, S.: A new approach to all-pairs shortest paths on real-weighted graphs.
Theor. Comput. Sci. 312(1), 47–74 (2004)
12. Sahni, S.: Data Structures, Algorithms, and Applications in C++. McGraw-Hill
Pub. Co., 1st edn. (1999)
13. Vassilevska Williams, V., Williams, R.: Subcubic equivalences between path, ma-
trix and triangle problems. In: FOCS 2010. pp. 645–654 (2010)
14. Yen, J.Y.: Finding the k shortest loopless paths in a network. Networks 17(11),
712–716 (1971)
| n   | Med | Q1  | Q2  | Q3  | Q99 | k=2000 | k=5000 | k=10000 | k=20000 |
|-----|-----|-----|-----|-----|-----|-------|--------|---------|---------|
| 2000| 0.22| 0.11| 0.10| 0.08| 0.02| 0.02  | 0.02   | 0.02    | 0.02    |
| 4000| 0.22| 0.11| 0.10| 0.08| 0.02| 0.02  | 0.02   | 0.02    | 0.02    |
| 8000| 0.22| 0.11| 0.10| 0.08| 0.02| 0.02  | 0.02   | 0.02    | 0.02    |

Table 6: Median and 90% quantile $Q_{99}$ for $n \in \{200, 500, 1000\}$. 

For $k \in (200, 500, 1000)$. 

13
A red square marks the mean, but was omitted for the SB algorithms as they would completely cover their corresponding boxes.

Fig. 2: Boxplots of running times in seconds for random graphs with 10,000 nodes and $k = 2000$. Plus signs represent outliers.
Fig. 3: Boxplots of running times in seconds for TIGER graphs. Plus signs represent outliers. A red square marks the mean. The interquartile range of SB on FLA for $k \in \{100, 200\}$ is so small that the boxes appear as lines.