

On the k -simple shortest paths problem in weighted directed graphs

Liam Roditty *

Abstract

We obtain the first approximation algorithm for finding the k -**simple** shortest paths connecting a pair of vertices in a weighted directed graph. Our algorithm is deterministic and has a running time of $O(k(m\sqrt{n} + n^{3/2} \log n))$, where m is the number of edges in the graph and n is the number of vertices. Let $s, t \in V$; the length of the i -th simple path from s to t computed by our algorithm is at most $\frac{3}{2}$ times the length of the i -th shortest simple path from s to t . The best algorithms for computing the exact k -simple shortest paths connecting a pair of vertices in a weighted directed graph are due to Yen [19] and Lawler [13]. The running time of their algorithms, using modern data structures, is $O(k(mn + n^2 \log n))$. Both algorithms are from the early 70's. Although this problem and other variants of the k -shortest path problem drew a lot of attention during the last three and a half decades, the $O(k(mn + n^2 \log n))$ bound is still unbeaten.

1 Introduction

Computing the shortest path connecting a pair of vertices in a graph is one of the most fundamental algorithmic problems in graph theory. A natural generalization of this problem is to compute a set of k -shortest paths that connect a given pair of vertices, for some $k > 1$. More formally, given a graph $G(V, E)$ with non-negative edge weights, a positive integer $k > 1$, and two vertices $s, t \in V$, the algorithm has to compute the k -shortest paths from s to t . Yen [19] and Lawler [13] showed how to compute the k -**simple** shortest paths in weighted directed graphs. The running time of their algorithms, when modern data structures are used, is $O(k(mn + n^2 \log n))$. For the restricted case of undirected graphs, Katoh, Ibaraki and Mine presented in [12] an algorithm with a running time of $O(k(m + n \log n))$, when modern data structures are used. In [9] Eppstein presented an algorithm, for weighted directed graphs, that finds the k -shortest paths when the paths are not required to be simple, i.e., they may contain loops. The running time of his algorithm is $O(m + n \log n + k)$. For unweighted directed graphs we presented in [17] a randomized algorithm with a running time of $O(m\sqrt{n} \log^2 n)$.

The algorithm of Yen goes back to the early 70's. Throughout the last three decades there were many improved implementations of Yen's algorithm [2, 10, 5, 15]. However, the bound of $O(k(mn + n^2 \log n))$, is still unbeaten.

The most classical and well-known shortest path problem is the problem of computing the All-Pairs Shortest-Paths (APSP) of a graph. For directed graphs with real edge weights the fastest known algorithm is due to Pettie [16]. Its running time is $O(mn + n^2 \log \log n)$. Many efforts have been made during the years to beat the $O(mn)$ time bound. Throughout the years very small asymptotic improvements were obtained and only when $m = n^2$; see [20] for more details. Recently, Chan [3] succeeded to get $o(mn)$ running time for unweighted undirected graphs. There is also a long line of papers [1, 7, 4, 8, 18] which present algorithms for approximating the APSP in undirected graphs. This type of algorithms obtains much faster running time by finding only approximated shortest paths and not the exact ones. For directed graphs, however, it is shown in [7] that using approximation cannot help in getting $o(mn)$ running time. Intuitively, it seems that finding k -simple shortest paths between two given vertices in a weighted directed graph should be done much faster than computing APSP. The fact that the undirected version of the problem can be solved in $O(k(m + n \log n))$ time also strengthens this feeling.

In this paper we present the first evidence that finding the k -simple shortest paths in a weighted directed graph might be an easier problem than finding APSP. We show that in this case the use of approximation does help. If we are willing to settle for approximated shortest paths, the running time can be improved significantly. In particular, we present a deterministic algorithm with a running time of $O(k(m\sqrt{n} + n^{3/2} \log n))$. The length of the i -th simple path from s to t computed by our algorithm is at most $\frac{3}{2}$ times more than the length of the i -th shortest simple path from s to t . We say that $\frac{3}{2}$ is the stretch of the path. The stretch of all the paths computed by our algorithm is at most $\frac{3}{2}$. Our algorithm is built upon an approximation algorithm for finding the second simple shortest path. This algorithm is based, among other things, on a novel technique for computing, in parallel, many truncated shortest paths trees at the cost of only a single run of Dijkstra's algorithm.

A closely related problem to the k -simple shortest paths

*School of Computer Science, Tel Aviv University, Tel Aviv 69978, Israel. E-mail address: liamr@cs.tau.ac.il

problem is the replacement paths problem. In the *replacement paths* problem we are required to find, for every edge e on the shortest path from s to t , a shortest path from s to t that avoids e , see Demetrescu and Thorup [6] for more details. In our context, it is quite easy to see that the second shortest path is one of the replacement paths. Many algorithms for computing k -shortest paths solve $O(k)$ instances of the replacement paths problem. In undirected graphs solving a replacement path problem has no effect on the asymptotic running time since there are algorithms whose running time is $O(m + n \log n)$; see Malik *et al.* [14] and Hershberger and Suri [11]. However, in weighted directed graphs nothing better than the trivial algorithm, which excludes each edge on the path in its turn and computes a shortest path from s to t , is known. The backbone of our algorithm, the approximation algorithm for the second shortest path, does not solve a replacement path problem.

Our algorithm for approximating the k -simple shortest path has the following useful property. The approximation of the i -th shortest path is obtained by finding another simple path in the graph of stretch $\frac{3}{2}$ with respect to the i -th shortest path. As a result of that if the length of the $i + 1$ -th shortest path is more than $\frac{3}{2}$ the length of the i -th shortest path then when computing k -simple shortest paths using our approximated algorithm, with $k \geq i$, the exact i -th shortest path is included in the output. In general, if the lengths of the k -simple shortest paths grow by factors of more than $\frac{3}{2}$, exact paths are found.

There is a numerous number of applications in which shortest paths computation is needed. Many of these applications need to compute a set of shortest paths and not only the shortest one. In [9] Eppstein lists a number of application for k -shortest paths computation. The first application mentioned there which also was the motivation for Lawler [13], is the case when a short path is needed but there are other constraints in addition to short length. These other constraints, however, are not well defined or are hard to optimize. A natural solution in such a case is to compute a set of shortest paths and to choose one of them by considering the other constraints. Our algorithm for finding approximated k -simple shortest paths can be used in such a situation, especially if we consider its property that if the lengths of the paths are well spread then the exact paths are found by it. For a comprehensive list of applications see [9].

The rest of this extended abstract is organized as follows. We start to present our techniques in Section 2 with a 2-approximation algorithm of the second shortest path. We then proceed to Section 3 and show how to improve the stretch factor to $\frac{3}{2}$. In Section 4 we present our approximation algorithm for the k -simple shortest paths. We end in Section 5 with some concluding remarks and

open problems.

2 A 2-Approximation of the second shortest path

In this section we present an algorithm that computes a 2-approximation of the second shortest path from s to t . Although our main result uses our stretch $\frac{3}{2}$ algorithm, we start with this, somewhat, simpler algorithm to explain ideas that will be used later on in a more complex way.

Let $G = (V, E)$ be a directed graph and let s and t be two vertices in the graph. Let $P(s, t) = \{s = u_1, u_2, \dots, u_q = t\}$ be a shortest path from s to t . Let $P^E(s, t) = \{(u_1, u_2), \dots, (u_{q-1}, u_q)\}$ be the set of edges of $P(s, t)$. For $u_i, u_j \in P(s, t)$, with $i < j$, let $P(u_i, u_j)$ denote the subpath of $P(s, t)$ from u_i to u_j . The objective of a second shortest path algorithm is to find a simple path from s to t whose length is minimal among all paths in the graph from s to t other than the shortest path from s to t . In the sequel, the i -th simple shortest path from s to t is denoted with $P_i(s, t)$. $P(s, t)$ always denotes the shortest path from s to t . Its number of vertices is denoted with $|P(s, t)|$. The length of $P_i(s, t)$ is denoted by ℓ_i and its approximated distance is denoted by $\hat{\ell}_i$.

Next, we define a detour:

DEFINITION 2.1. (A DETOUR) *Let $P(s, t)$ be a simple path from s to t . A simple path from u to v , $D(u, v)$, is a detour of $P(s, t)$ if $D(u, v) \cap P(s, t) = \{u, v\}$ and u precedes v on $P(s, t)$.*

It is easy to see that a second shortest path from s to t is a path composed from three parts: an initial part on $P(s, t)$ from s to some vertex u_i , a detour from u_i to some vertex u_j , where $j > i$, and a final part on $P(s, t)$ from u_j to t . We denote such a path by $P(s, u_i) \cdot D(u_i, u_j) \cdot P(u_j, t)$, where ‘ \cdot ’ stands for concatenating two paths.

In [17] the replacement paths in directed unweighted graphs were found by dividing the detours into long and short detours according to the number of edges. Since the graph was unweighted there was a clear distinction between the detours. However, in weighted graphs there is no relation between the number of edges a detour has to its actual length, and other ideas should be used. Recall that our target is to get an algorithm with a running time of $\tilde{O}(m\sqrt{n})^1$, thus, as long as $|P(s, t)| \leq \sqrt{n}$, a trivial algorithm which, excludes every edge from the shortest path in its turn, computes a shortest path without it and then after doing it for every edge on the path picks the best path among all the paths computed, will find the exact second shortest path in $\tilde{O}(m\sqrt{n})$ time. The problem becomes much harder when $|P(s, t)| > \sqrt{n}$. Our

¹ \tilde{O} hides a factor of $O(\text{polylog}(n))$

```

algorithm UpperBound( $G(V, E)$ )
 $W \leftarrow \infty$ 
for  $i \leftarrow 1$  to  $\frac{|P(s, t)|}{\sqrt{n}}$ 
   $E_i \leftarrow \{(u_j, u_{j+1}) \mid (i-1)\sqrt{n} + 1 \leq j \leq i\sqrt{n}\}$ 
   $W \leftarrow \min(W, \text{Dijkstra}(G(V, E \setminus E_i), s, t))$ 
return  $W$ 

algorithm SetParameters( $G(V, E)$ )
for  $i \leftarrow 1$  to  $|P(s, t)|$ 
   $h(u_i) \leftarrow \frac{W}{2} - d[s, u_i] - d[u_{i+2\sqrt{n}}, t]$ 
  if  $i \leq 2\sqrt{n}$  then  $p(u_i) \leftarrow 0$ 
  else  $p(u_i) \leftarrow h(u_{i-2\sqrt{n}}) + p(u_{i-2\sqrt{n}})$ 
   $S_{i \% 2\sqrt{n}} \leftarrow S_{i \% 2\sqrt{n}} \cup \{u_i\}$ 

algorithm ParallelDijkstra( $G(V, E), S$ )
for every  $v \in V \setminus S$  do Insert( $H, v, \infty$ )
for every  $(s, v) \in E$  s.t.  $(s, v) \in S \times V$  do
  Relax( $(s, v), s$ )
while  $H \neq \phi$  do
   $u \leftarrow \text{Extract-Min}(H)$ 
   $\bar{d}[src(u), u] = d[u]$ 
  for every  $(u, v) \in E \setminus P^E(s, t)$  do
    if  $v \in H$  then Relax( $(u, v), src(u)$ )

algorithm Relax( $(u, v), s$ )
 $x \leftarrow \bar{d}[s, u] + \text{weight}(u, v)$ 
if  $(x < h(s)) \wedge (x + p(s) \leq d[v] + p(src(v)))$  then
   $d[v] \leftarrow x$ 
   $src(v) \leftarrow s$ 
  Decrease-Key( $H, v, d[v] + p(s)$ )

```

Figure 1: The procedure for computing the upper bound and for setting the parameters are in the first row. The parallel implementation of Dijkstra and its corresponding relax procedure are in the second row.

algorithm computes as a first step an upper bound on ℓ_2 . This upper bound is then used to synchronize many runs of Dijkstra's algorithm so that their total cost will be $O(m + n \log n)$, as the cost of a single run. In the parallel run of Dijkstra we compute truncated shortest paths trees for a set of vertices of the path $P(s, t)$. We repeat on this parallel run until we have truncated shortest paths trees for every vertex of $P(s, t)$. The information gathered in these trees is used to either improve the upper bound on ℓ_2 by finding a better detour or to prove that the upper bound gives an approximation of 2.

The upper bound is found using a method which is also used, in a similar context, by Demetrescu and Thorup

in [6]. The algorithm is given in Figure 1. To simplify the presentation and to avoid rounding problems, we assume that the number of vertices $P(s, t)$ has, is a multiple of $2\sqrt{n}$, which in its turn is an integer. However, dealing with paths with arbitrary number of vertices is straightforward. The algorithm performs at most \sqrt{n} iterations and in each iteration a different set of \sqrt{n} consecutive edges is removed from the graph and a shortest path from s to t is computed. The length of the shortest, among all the paths computed, is the value of the upper bound. We denote this value with W throughout the paper. The running time of the algorithm is $O(\sqrt{n}(m + n \log n))$. If $W \leq 2\ell_1$ then W is already a 2-approximation of the second shortest path length since $\ell_1 \leq \ell_2 \leq W \leq 2\ell_1$ and the algorithm simply returns W . Thus, throughout this section we assume that $W > 2\ell_1$.

The next Lemma, which is a direct result of the way the upper bound is computed, is essential to the correctness of the next stages of the algorithm.

LEMMA 2.1. *The length of any path $P(s, u_i) \cdot D(u_i, u_j) \cdot P(u_j, t)$, where $j - i \geq 2\sqrt{n}$, is at least W .*

Proof. Since $j - i \geq 2\sqrt{n}$ there exists an integer p such that $i \leq (p-1) \cdot \sqrt{n} + 1$ and $p \cdot \sqrt{n} \leq j$. While computing the upper bound the set E_p was excluded from the graph and the path $P(s, u_i) \cdot D(u_i, u_j) \cdot P(u_j, t)$ was considered. Thus, its length must be at least W , otherwise, the upper bound would have been smaller than W .

An important corollary of this Lemma is the following.

COROLLARY 2.1. *If the length of the path $P(s, u_i) \cdot D(u_i, u_j) \cdot P(u_j, t)$ is strictly less than W then $j - i$ must be strictly less than $2\sqrt{n}$.*

After computing the upper bound we set for every $u_i \in P(s, t)$ its priority $p(u_i)$ and the maximal height of a truncated shortest paths tree rooted at it $h(u_i)$ (see Figure 1). Note that $h(u_i)$ is always positive since $W > 2d[s, t]$. We then proceed by dividing the vertices of the path into sets. We will have a total number of $2\sqrt{n}$ sets, each contains at most $\frac{|P(s, t)|}{2\sqrt{n}}$ vertices. (Recall that for simplicity we assume that $|P(s, t)|$ is a multiple of $2\sqrt{n}$). For $j \in [1, 2\sqrt{n}]$ let $S_j = \{u_{j+2\sqrt{n} \cdot i} \mid 0 \leq i < |P(s, t)|/2\sqrt{n}\}$ be the j -th set. It is crucial for the algorithm that any two vertices of the path which are in the same set are separated by at least $2\sqrt{n} - 1$ vertices of $P(s, t)$. The parallel Dijkstra and its corresponding relax procedure are given in Figure 1. Let S be a set as defined above. In a single run the parallel Dijkstra algorithm computes for every $u \in S$ a shortest paths tree up to depth $h(u)$. It starts by relaxing all the edges going out from vertices of S . For every $v \in V \setminus S$ the algorithm keeps its current source, i.e., the vertex from S which currently

responsible for the estimated distance $d[v]$, in an array named src . Let u be a vertex which is extracted from the heap then $\bar{d}[src(u), u]$ is set to $d[u]$ (\bar{d} denotes distances in the graph $G(V, E \setminus P^E(s, t))$) and its outgoing edges are relaxed if their head is still in the heap. Notice that the edges $P^E(s, t)$ are never relaxed and the parallel Dijkstra ignores them. The usual relax procedure is modified and a vertex key is decreased only if it satisfies the following two constraints. The first, called the height constraint, is that the length of the new path does not diverge from the height of the truncated tree of its source. The second, called the priority constraint, is that the sum of the length of the new path and the priority of its source is less than the sum of the current estimation of the distance and the priority of the current source. Notice that, in contrast to the usual Dijkstra algorithm, the key of a vertex is not $d[v]$ but $d[v] + p(s)$. As we will see shortly, this is done in order to prioritize the paths, and to prevent ‘vertex stealing’ from one truncated tree to another.

We need to prove that the running time of the parallel Dijkstra is $O(m + n \log n)$. The edges outgoing from vertices of the set S are obviously relaxed only once. Any other edge is also relaxed once when its tail is extracted from H . It stems from the fact that, once a vertex is extracted from H , it is never re-inserted. Thus, the running time is $O(m + n \log n)$.

We turn our attention to prove an important property of the parallel Dijkstra. This property is used later on to establish the correctness proof of the algorithm.

LEMMA 2.2. *Let $P(s, t)$ be the shortest path from s to t and let W be the upper bound. Let $P(s, u_i) \cdot D(u_i, u_j) \cdot P(u_j, t)$ be a second simple shortest path whose length is less than $\frac{W}{2}$. The truncated tree computed for u_i by the parallel Dijkstra algorithm contains the path $D(u_i, u_j)$.*

Proof. From Corollary 2.1 it follows that such a ‘short’ path must satisfy that $j - i < 2\sqrt{n}$, otherwise W would not be the upper bound. Let S be the set in which u_i is contained and let h be the length of $D(u_i, u_j)$. We know that $h < \frac{W}{2} - d[s, u_i] - d[u_j, t]$. The height of a tree computed for u_i by the parallel Dijkstra is at most $h[u_i]$, where $h[u_i] = \frac{W}{2} - d[s, u_i] - d[u_{i+2\sqrt{n}}, t]$. It is easy to see that $h < h[u_i]$ since $j < i + 2\sqrt{n}$ and $d[u_{i+2\sqrt{n}}, t] \leq d[u_j, t]$. Thus, every vertex of the path $D(u_i, u_j)$ can be part of the truncated tree of u_i from the perspective of the height constraint. We show that the path $D(u_i, u_j)$ is, indeed, contained in the truncated tree of u_i .

Assume it is not, and let $y \in D(u_i, u_j)$ be the first vertex of $D(u_i, u_j)$ such that $src(y) = u_k$, for some $u_k \in S \setminus \{u_i\}$, when the execution of the parallel Dijkstra halts (see Figure 2). Thus, y is in the truncated tree of u_k . Let $D(u_k, y)$ be the path from u_k to y in the truncated tree

of u_k . We divide the proof into two cases. First we show that k cannot be strictly smaller than i and then we show that k cannot be strictly larger than i .

Assume $k < i$. We examine the path created by the following concatenation of paths, $P(s, u_k) \cdot D(u_k, y) \cdot D(y, u_j) \cdot P(u_j, t)$. The path $D(u_k, y)$ satisfies the height constraint of u_k . The path $D(y, u_j)$ is a portion of the path $D(u_i, u_j)$ which satisfies the height constraint of u_i . Adding it together we get that there is a simple path from s to t whose length is strictly less than $d[s, u_k] + h(u_k) + h(u_i) + d[u_j, t]$. Replacing $h(u_i)$ and $h(u_k)$ with their values we get:

$$\begin{aligned} & d[s, u_k] + \frac{W}{2} - d[s, u_k] - d[u_{k+2\sqrt{n}}, t] + \frac{W}{2} \\ & \quad - d[s, u_i] - d[u_{i+2\sqrt{n}}, t] + d[u_j, t] = \\ & = W - d[s, u_i] - d[u_{i+2\sqrt{n}}, t] - d[u_{k+2\sqrt{n}}, t] + d[u_j, t] \end{aligned}$$

We bound the value of this expression with W . Recall that $u_k, u_i \in S$, thus, $k + 2\sqrt{n} \leq i$. Also note that $j > i$, otherwise the path is not simple. Hence, $k + 2\sqrt{n} < j$ and $d[u_{k+2\sqrt{n}}, t] \geq d[u_j, t]$. We get that:

$$W - d[s, u_i] - d[u_{i+2\sqrt{n}}, t] - d[u_{k+2\sqrt{n}}, t] + d[u_j, t] \leq W$$

We conclude that the length of the path $P(s, u_k) \cdot D(u_k, y) \cdot D(y, u_j) \cdot P(u_j, t)$ is strictly less than W . This leads to a contradiction from the following reason: Between u_k and u_j there are at least $2\sqrt{n}$ vertices and by Lemma 2.1 the length of such a path must be at least W , a contradiction.

Assume $k > i$. The key of every vertex that appears in the truncated tree of u_k when it is extracted from the heap is at least $p(u_k)$. Thus, when y is extracted its key is at least $p(u_k)$. Let x_1, x_2, \dots, x_q be the vertices on the path $D(u_i, u_j)$ between u_i and y , if any (it might be that $q = 0$). The key of every x_i , when it is extracted from the heap, is strictly less than $p(u_k)$. The edge (u_i, x_1) is relaxed before any vertex is extracted, thus, x_1 is extracted before y . When it is extracted the edge (x_1, x_2) is relaxed. Similarly, all the vertices, including x_q , are extracted before y . When x_q is extracted the edge (x_q, y) is relaxed and y 's key is decreased and its source is changed (note that if $q = 0$ then the edge (u_i, y) is relaxed as it is an outgoing edge of u_i). This contradicts the fact that y is in the truncated tree of u_k .

From the two cases analyzed above it follows that the path $D(u_i, u_j)$ is contained in the truncated tree of u_i .

The algorithm for computing a 2-approximation of the second shortest path is given in Figure 3. The algorithm performs $2\sqrt{n}$ iterations. In each iteration the parallel Dijkstra is called with a different set of vertices. The algorithm computes a truncated shortest paths tree for every vertex of $P(s, t)$. After every execution of the

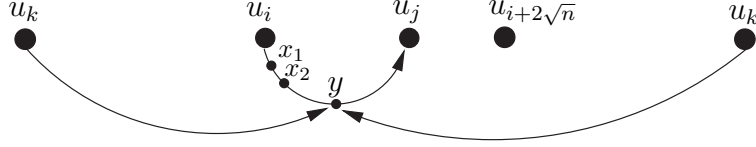


Figure 2: The possible paths and their respective sources.

algorithm 2-SecondPath($G(V, E), P(s, t)$)

```

 $W \leftarrow \text{UpperBound}(G(V, E))$ 
 $\hat{\ell}_2 \leftarrow W$ 
if  $\hat{\ell}_2 \leq 2\ell_1$  then return  $\hat{\ell}_2$ 
SetParameters( $G(V, E)$ )
for  $j \leftarrow 1$  to  $2\sqrt{n}$ 
   $\bar{d} \leftarrow \text{ParallelDijkstra}(G(V, E), S_j)$ 
  for every  $u_p \in S_j$ 
    for  $i \leftarrow p + 1$  to  $p + 2\sqrt{n} - 1$  do
      if  $\hat{\ell}_2 > d[s, u_p] + \bar{d}[u_p, u_i] + d[u_i, t]$  then
         $\hat{\ell}_2 \leftarrow d[s, u_p] + \bar{d}[u_p, u_i] + d[u_i, t]$ 
return  $\hat{\ell}_2$ 

```

Figure 3: A 2-approximation of the second shortest path

parallel Dijkstra the algorithm tries to improve the best second shortest path it currently has.

The next Theorem summarizes the main result of this section.

THEOREM 2.1. *The algorithm for computing a 2-approximation of the second shortest path given in Figure 3 computes a simple path of length at most twice the length of the simple second shortest path from s to t . The running time of the algorithm is $O(\sqrt{n}(m + n \log n))$.*

Proof. Let W be the upper bound initially computed by the upper bound algorithm. Let ℓ_2 be the exact length of the second shortest path. If $\ell_2 = W$ then the exact length is computed. On the other hand, if $\ell_2 < \frac{W}{2}$ then by Lemma 2.2, our algorithm detects this path and again the exact length is computed. Finally, if $\frac{W}{2} \leq \ell_2 < W$ the algorithm returns W which is at most twice the length of the exact second shortest path. The algorithm performs $O(\sqrt{n})$ iterations of Dijkstra and parallel Dijkstra, both at a cost of $O(m + n \log n)$.

3 A $\frac{3}{2}$ -Approximation of the second shortest path

In this section we show how to improve the stretch factor to $\frac{3}{2}$. As before we compute an upper bound using the upper bound algorithm. By Lemma 2.1 we know that the

algorithm SetParameters($G(V, E), dr$)

```

for  $i \leftarrow 1$  to  $|P(s, t)|$ 
   $h(u_i) \leftarrow \max(\frac{W}{3} - dr \cdot d[s, u_i] - (1 - dr) \cdot d[u_i, t], 0)$ 
  if  $i \leq 2\sqrt{n}$  then
     $p(u_i) \leftarrow 0$ 
  else
     $p(u_i) \leftarrow h(u_{i-2\sqrt{n}}) + p(u_{i-2\sqrt{n}})$ 
   $S_{i \% 2\sqrt{n}} \leftarrow S_{i \% 2\sqrt{n}} \cup \{u_i\}$ 

```

algorithm $\frac{3}{2}$ SecondPath($G(V, E), P(s, t)$)

```

 $W \leftarrow \text{UpperBound}(G(V, E))$ 
 $\hat{\ell}_2 \leftarrow W$ 
if  $\hat{\ell}_2 \leq \frac{3}{2}\ell_1$  then return  $\hat{\ell}_2$ 
SetParameters( $G(V, E), 1$ )
for  $j \leftarrow 1$  to  $2\sqrt{n}$ 
   $\bar{d} \leftarrow \text{ParallelDijkstra}(G(V, E), S_j)$ 
SetParameters( $G(V, E), 0$ )
for  $j \leftarrow 1$  to  $2\sqrt{n}$ 
   $\bar{d} \leftarrow \text{ParallelDijkstra}(\overleftarrow{G}(V, E), S_j)$ 
return BestPath( $G(V, E)$ )

```

Figure 4: A 1.5-approximation of the second shortest path.

length of any path of the form $P(s, u_i) \cdot D(u_i, u_j) \cdot P(u_j, t)$, where $j - i \geq 2\sqrt{n}$, is at least W . Thus, the key for getting a better approximation is to detect paths whose length is more than $\frac{W}{2}$ and smaller than W in the $O(\sqrt{n})$ iterations of the parallel Dijkstra algorithm. We show that this is indeed possible. In this section we show that by applying more complex techniques any path of length less than $\frac{2W}{3}$ can be found. The asymptotic running time, however, is not effected.

The algorithm is composed from the following steps: In the first step it computes a truncated shortest paths tree using the parallel Dijkstra for every $u_i \in P(s, t)$. We denote such a tree by $T_{\text{out}}(u_i)$. The height constraint of $T_{\text{out}}(u_i)$ is $\max(\frac{W}{3} - d[s, u_i], 0)$. In the second step the edges of the graph are reversed and a truncated shortest paths tree is computed, once again, for every $u_i \in P(s, t)$.

We denote such a tree by $T_{\text{in}}(u_i)$. The height constraint of $T_{\text{in}}(u_i)$ is $\max(\frac{W}{3} - d[u_i, t], 0)$. In the third step the algorithm tries to find a better approximation for the second shortest path by concatenating paths. The algorithm is given in Figure 4. The procedure for setting the parameters of the vertices of $P(s, t)$ is also given in Figure 4. Note that in order to support the two directions computation an additional value dr is passed to the procedure. When the edges are in their original direction dr is set to 1 and when they are reversed dr is set to 0. Recall that \bar{d} denotes distances in the graph $G(V, E \setminus P^E(s, t))$.

In the next Lemma we prove a useful property of detours of paths whose length is less than $\frac{2W}{3}$.

LEMMA 3.1. *Let $P(s, u_i) \cdot D(u_i, u_j) \cdot P(u_j, t)$ be a simple path whose length is less than $\frac{2W}{3}$. There must be at least one edge (u, v) on $D(u_i, u_j)$ such that u satisfies the height constraint of u_i in $G(V, E)$ and v satisfies the height constraint of u_j when the edges of the graph are reversed.*

Proof. Assume first that $d[s, u_i] < \frac{W}{3}$ and $d[u_j, t] < \frac{W}{3}$. We show that there is an edge (u, v) on $D(u_i, u_j)$ such that $\bar{d}[u_i, u] < \frac{W}{3} - d[s, u_i]$ and $\bar{d}[v, u_j] < \frac{W}{3} - d[u_j, t]$. Let $u \in D(u_i, u_j)$ be the farthest vertex from u_i which satisfies $\bar{d}[u_i, u] < \frac{W}{3} - d[s, u_i]$. If $u = u_j$ then it is easy to see that the last edge of $D(u_i, u_j)$, i.e., the one that goes into u_j , satisfies the requirements, thus we can assume that $u \neq u_j$. Let (u, v) be the edge outgoing from u on the path $D(u_i, u_j)$. We prove that $\bar{d}[v, u_j] < \frac{W}{3} - d[u_j, t]$. The length of $P(s, u_i) \cdot D(u_i, u_j) \cdot P(u_j, t)$ equals $d[s, u_i] + \bar{d}[u_i, v] + \bar{d}[v, u_j] + d[u_j, t]$ and is smaller than $\frac{2W}{3}$. Thus,

$$\bar{d}[v, u_j] < \frac{2W}{3} - d[s, u_i] - \bar{d}[u_i, v] - d[u_j, t]$$

Since v does not satisfy the height constraint of u_i it follows that $\bar{d}[u_i, v] \geq \frac{W}{3} - d[s, u_i]$. This implies that $\bar{d}[v, u_j] < \frac{W}{3} - d[u_j, t]$ as desired.

Assume now that $d[s, u_i] \geq \frac{W}{3}$, i.e., the truncated tree of u_i contains only itself. Let (u_i, v) be the first edge on the path $D(u_i, u_j)$. The length of the path $D(u_i, u_j) \cdot P(u_j, t)$ is less than $\frac{W}{3}$ and hence, $\bar{d}[v, u_j] < \frac{W}{3} - d[u_j, t]$. The case that $d[u_j, t] \geq \frac{W}{3}$ is identical.

In the sequel, we call an edge that satisfies the conditions of Lemma 3.1 a border edge. In the next Lemma we show that not only that the endpoints of a border edge satisfy the height constraints but they are also part of the suitable truncated trees.

LEMMA 3.2. *Let $P(s, u_i) \cdot D(u_i, u_j) \cdot P(u_j, t)$ be a simple path whose length is less than $\frac{2W}{3}$. If $(u, v) \in E$ is*

a border edge of $D(u_i, u_j)$ then $u \in T_{\text{out}}(u_i)$ and $v \in T_{\text{in}}(u_j)$.

Proof. From Lemma 3.1 it follows that the vertex u satisfies the height constraint of u_i . Thus, every vertex of the path $D(u_i, u)$ can be part of the truncated tree of u_i from the perspective of the height constraint. Let $u_i \in S$. We show that the path $D(u_i, u)$ is, indeed, contained in the truncated tree computed for u_i by the parallel Dijkstra when it is executed with S .

Assume it is not and let $y \in D(u_i, u)$ be the first vertex of $D(u_i, u)$ such that $\text{src}(y) = u_k$, for some $u_k \in S \setminus \{u_i\}$, when the execution of the parallel Dijkstra halts. Let $D(u_k, y)$ be the path discovered by the algorithm. As before we divide the proof into two cases. First we show that k cannot be strictly smaller than i and then we show that k cannot be strictly larger than i .

Assume $k < i$. We examine the path created by the following concatenation of paths, $P(s, u_k) \cdot D(u_k, y) \cdot D(y, u_j) \cdot P(u_j, t)$. The path $D(u_k, y)$ satisfies the height constraint of u_k . The path $D(y, u_j)$ is a portion of the path $D(u_i, u_j)$, thus, we can bound the length of $D(y, u_j)$ with $\frac{2W}{3} - d[s, u_i] - d[u_j, t]$. Adding it all together we get that there is a simple path from s to t whose length is strictly less than:

$$\begin{aligned} d[s, u_k] + h(u_k) + \frac{2W}{3} - d[s, u_i] - d[u_j, t] + d[u_j, t] = \\ d[s, u_k] + \frac{W}{3} - d[s, u_k] + \frac{2W}{3} - d[s, u_i] = W - d[s, u_i] \end{aligned}$$

This leads to a contradiction from the following reason: $j > i$ and both u_k and u_i are in the same set S which implies that $i - k \geq 2\sqrt{n}$. Thus, between u_k and u_j there are at least $2\sqrt{n}$ vertices. By Lemma 2.1 we know that the length of such a path must be at least W .

Assume $k > i$. This case is identical to the one in Lemma 2.2.

We conclude that $D(u_i, u)$ is in the truncated tree of u_i . The proof that $v \in T_{\text{in}}(u_j)$ is symmetrical.

Let $P(s, u_i) \cdot D(u_i, u_j) \cdot P(u_j, t)$, where $j - i < 2\sqrt{n}$, be a simple path whose length is less than $\frac{2W}{3}$ and let $(u, v) \in D(u_i, u_j)$ be a border edge. According to Lemma 3.2 the path $D(u_i, u_j)$ can be partitioned into three parts: $D(u_i, u)$, (u, v) and $D(v, u_j)$, such that, $u \in T_{\text{out}}(u_i)$ and $v \in T_{\text{in}}(u_j)$. In the rest of this section we explain how to use the truncated trees computed in order to find these paths in $\tilde{O}(m\sqrt{n})$ time. The main difficulty that we are facing is that a vertex may participate in $\Omega(\sqrt{n})$ truncated trees. However, as we show in the next Lemma, in our search for a better path, it is sufficient to consider only $2\sqrt{n}$ trees.

algorithm BestPath($G(V, E)$)

```

for every  $u \in V$  do Init( $u$ )
for every  $(u, v) \in E \setminus P^E(s, t)$ 
  for  $i \leftarrow \tau(u) + 1$  to  $\tau(u) + 2\sqrt{n} - 1$ 
    if  $\hat{\ell}_2 > \min(H_{\text{in}}(u)) + w(u, v) + \min(H_{\text{out}}(v))$ 
       $\hat{\ell}_2 \leftarrow \min(H_{\text{in}}(u)) + w(u, v) + \min(H_{\text{out}}(v))$ 
      Delete( $H_{\text{out}}(v), u_i$ )
      Insert( $H_{\text{in}}(u), u_i, d[s, u_i] + \bar{d}[u_i, u]$ )
      Init( $u$ ), Init( $v$ )
return  $\hat{\ell}_2$ 

```

algorithm Init(u)

```

 $H_{\text{in}}(u) \leftarrow \phi, H_{\text{out}}(u) \leftarrow \phi$ 
Insert( $H_{\text{in}}(u), u_{\tau(u)}, d[s, u_{\tau(u)}] + \bar{d}[u_{\tau(u)}, u]$ )
for  $i \leftarrow \tau(u) + 1$  to  $\tau(u) + 2\sqrt{n} - 1$ 
  Insert( $H_{\text{out}}(u), u_i, d[u, u_i] + \bar{d}[u_i, t]$ )

```

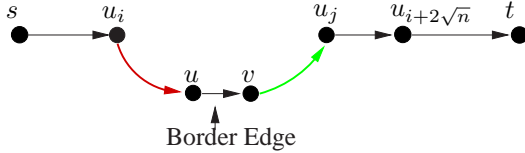


Figure 5: The procedure for finding a better path and an example of such a path

LEMMA 3.3. *Let $\tau(u)$ be the smallest index such that $u \in T_{\text{out}}(u_{\tau(u)})$ and let $u \in T_{\text{out}}(u_i)$ for some $i \geq \tau(u) + 2\sqrt{n}$. Any path of the form $P(s, u_i) \cdot D(u_i, u) \cdot (u, v) \cdot D(v, u_j) \cdot P(u_j, t)$ is of length at least $\frac{2W}{3}$.*

Proof. Assume, for the sake of contradiction, that there is a path $P(s, u_i) \cdot D(u_i, u) \cdot (u, v) \cdot D(v, u_j) \cdot P(u_j, t)$ whose length is strictly less than $\frac{2W}{3}$. The length of the path $P(s, u_{\tau(u)}) \cdot D(u_{\tau(u)}, u) \cdot (u, v) \cdot D(v, u_j) \cdot P(u_j, t)$ can be bounded as follows. By the height constraint of $u_{\tau(u)}$ we can bound the length of the portion $P(s, u_{\tau(u)}) \cdot D(u_{\tau(u)}, u)$ with $\frac{W}{3}$. The length of the portion $(u, v) \cdot D(v, u_j) \cdot P(u_j, t)$ is smaller than $\frac{2W}{3} - d[s, u_i]$. Adding it together we get that the length of the path is less than $W - d[s, u_i]$. Between $u_{\tau(u)}$ and u_j there are at least $2\sqrt{n}$ vertices and by Lemma 2.1 it is impossible that such a path has a length smaller than W .

The algorithm for searching for a better path is given in Figure 5. For every $u \in V$ two heaps, $H_{\text{in}}(u)$ and $H_{\text{out}}(u)$, are maintained. As follows from Lemma 3.3 only vertices from $P(u_{\tau(u)}, u_{\tau(u)+2\sqrt{n}-1})$ should be considered by u . When the heaps are initialized $H_{\text{in}}(u)$ contains the vertex $u_{\tau(u)}$ and $H_{\text{out}}(u)$ contains the vertices $u_{\tau(u)+1}, \dots, u_{\tau(u)+2\sqrt{n}-1}$. The key of u_i in $H_{\text{in}}(u)$ is

$d[s, u_i] + \bar{d}[u_i, u]$ and in $H_{\text{out}}(u)$ is $\bar{d}[u, u_i] + d[u_i, t]$. The initialization procedure is given also in Figure 5.

The algorithm processes every edge $(u, v) \in E \setminus P^E(s, t)$ in the following way. A path is built by adding the minimum from $H_{\text{in}}(u)$ and from $H_{\text{out}}(v)$, to the weight of (u, v) . If this path improves the current upper bound then the upper bound is updated with the length of the new path. The algorithm proceeds by deleting $u_{\tau(u)+1}$ from $H_{\text{out}}(v)$ and inserting it to $H_{\text{in}}(u)$. This procedure is done until $H_{\text{out}}(v)$ becomes empty. The endpoints u and v are then re-initialized and a new edge is picked. The total running time of this algorithm is $\tilde{O}(m\sqrt{n})$.

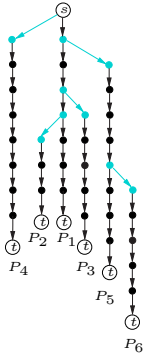
The next Theorem summarizes the main result of this section.

THEOREM 3.1. *The algorithm given in Figure 4 finds a path whose stretch is at most $\frac{3}{2}$ with respect to the length of the second simple shortest path from s to t . The running time of the algorithm is $\tilde{O}(m\sqrt{n})$.*

Proof. Let $P(s, u_i) \cdot D(u_i, u_j) \cdot P(u_j, t)$ be the exact simple second shortest path and let ℓ_2 be its length. If $\ell_2 = W$ then the algorithm computes the exact length. If $\ell_2 < \frac{2W}{3}$ then by Corollary 2.1 it must be that $j - i < 2\sqrt{n}$. Let (u, v) be a border edge of $D(u_i, u_j)$. From Lemma 3.2 it follows that $u \in T_{\text{out}}(u_i)$ and $v \in T_{\text{in}}(u_j)$. From Lemma 3.3 it follows that i must be smaller than $\tau(u) + 2\sqrt{n}$. Thus, when the edge (u, v) is considered there is a stage in which $u_i \in H_{\text{in}}(u)$ and $u_j \in H_{\text{out}}(v)$, and as a result of that, the value $\min(H_{\text{in}}(u)) + w(u, v) + \min(H_{\text{out}}(v))$, computed at this stage, equals to ℓ_2 and once again the exact length is computed. If $\frac{2W}{3} \leq \ell_2 < W$ the algorithm returns W and the stretch of the path is at most $\frac{3}{2}$. The parallel Dijkstra and Dijkstra are executed $O(\sqrt{n})$ times. The algorithm for finding the best path does $O(\sqrt{n} \log n)$ work per an edge. Thus, the total running time is $\tilde{O}(m\sqrt{n})$.

4 Finding the k -th simple shortest paths

In this section we show how to compute an approximation of the k -simple shortest paths. Our algorithm guarantees a stretch of $\frac{3}{2}$, for the i -th path computed by it, with respect to the i -th shortest path. In [17] we showed that computing the k -simple shortest paths can be done by $O(k)$ computations of second shortest paths. However, it is not straightforward to apply the algorithm from [17] when only an approximation algorithm of the second shortest path is available. We start this section with an overview of the k -simple shortest paths algorithm from [17]. We then proceed and explain how to modify the algorithm to work when only an approximation of the second shortest path can be computed. We end with a correctness proof for the modified algorithm. We assume that the second shortest path algorithm outputs a path



algorithm **k-SimplePath**($G(V, E), s, t, k$)

$P_1(s, t) \leftarrow \text{Dijkstra}(G(V, E), s, t)$

$T \leftarrow P_1(s, t)$

$E_d(s) \leftarrow \phi$

Insert($H, \langle \text{SecondPath}(G(V, E), P_1(s, t)), 1 \rangle$)

for $i \leftarrow 2$ to k

$\langle P_i(s, t), j \rangle \leftarrow \min(H)$

Concatenate $P_i(v_i, t)$ to u_i in T

$E_d(v_j) \leftarrow E_d(v_j) \cup \{(u_i, v_i)\}$

$E_d(v_i) \leftarrow \phi$

$P_i^2(v_i, t) \leftarrow \text{SecondPath}(G(V \setminus P_i(s, u_i), E \setminus E_d(v_i)), P_i(v_i, t))$

$P_j^2(v_j, t) \leftarrow \text{SecondPath}(G(V \setminus P_j(s, u_j), E \setminus E_d(v_j)), P_j(v_j, t))$

Insert($H, \langle P_i(s, v_i) \cdot P_i^2(v_i, t), i \rangle$)

Insert($H, \langle P_j(s, v_j) \cdot P_j^2(v_j, t), j \rangle$)

return T

Figure 6: A deviations tree and the k -simple shortest paths algorithm

and not only a length. Our algorithm can support that in the same asymptotic running time.

Let $P_i(s, t)$ be the i -th shortest path in G . The output of the k -simple shortest paths algorithm is given in the form of a tree of paths which we define as follows.

DEFINITION 4.1. (DEVIATION TREE) For $k = 1$ the tree is simply $P_1(s, t)$. Let T_{i-1} be the tree that contains the $i - 1$ -th shortest paths and let $P_i(s, t) = \{s, u_1, \dots, u_l, t\}$ be the i -th shortest path. Let $P_i(s, u_j)$ be the longest subpath of $P_i(s, t)$ already in T_{i-1} . We make a copy from $P_i(u_{j+1}, t)$ and concatenate it with the copy of $P_i(s, u_j)$ in T_{i-1} . The resulting tree is T_i . The edge (u_j, u_{j+1}) is said to be the deviation edge of $P_i(s, t)$.

The algorithm for computing the k -simple shortest paths is given in Figure 6. The algorithm maintains a heap H of paths. The algorithm performs k iterations and in each iteration the item with the minimal key is extracted from H . When the i -th iteration begins, every item of H is a pair $\langle Q, j \rangle$, where Q is a path and $j < i$. The key of the item is the length of Q . The path Q and the index j satisfy the following relation. Let (u_j, v_j) be the deviation edge of $P_j(s, t)$ and let $E_d(v_j)$ be the set of deviation edges emanate from vertices of the path $P_j(v_j, t)$ so far. (The purpose of the set $E_d(v_j)$ is to prevent from a path to be the second shortest path of $P_j(v_j, t)$ more than once.) In the graph $G(V \setminus P_j(s, u_j), E \setminus E_d(v_j))$ the path $P_j(v_j, t)$ is the shortest path from v_j to t and $Q(v_j, t)$ is the second shortest path from v_j to t . The portion $Q(s, v_j)$ of Q is simply $P_j(s, v_j)$.

In the i -th iteration the algorithm extracts a path from H . In [17] we proved that the extracted path is the i -th shortest path, $P_i(s, t)$. Let (u_i, v_i) be the deviation

edge of $P_i(s, t)$. After the extraction the algorithm computes a second shortest path $P_i^2(v_i, t)$ for $P_i(v_i, t)$ in the graph $G(V \setminus P_i(s, u_i), E \setminus E_d(v_i))$ and a second shortest path $P_j^2(v_j, t)$ for $P_j(v_j, t)$ in the graph $G(V \setminus P_j(s, u_j), E \setminus E_d(v_j))$. Notice that the edge (u_i, v_i) is added to $E_d(v_j)$ to avoid from the path $P_i(v_j, t)$ to be the second shortest path of $P_j(v_j, t)$. The algorithm inserts into H the following two pairs: $\langle P_i(s, v_i) \cdot P_i^2(v_i, t), i \rangle$ and $\langle P_j(s, v_j) \cdot P_j^2(v_j, t), j \rangle$.

The full details and proofs appear in [17] and are omitted from this extended abstract. The main issue that we need to address here, however, is the effect of using an approximation of the second shortest path.

We modify the algorithm in the following way. Let $\hat{P}_i^2(v_i, t)$ be the approximated second shortest path of $P_i(v_i, t)$ in the graph $G(V \setminus P_i(s, u_i), E \setminus E_d(v_i))$ and let (u, v) be the edge of $\hat{P}_i^2(v_i, t)$ which deviates from $P_i(v_i, t)$. We compute the shortest path from v to t in the graph $G(V \setminus P_i(s, u), E)$. Let $Q(v, t)$ be the resulting shortest path from v to t . The path inserted, eventually, to the heap is the following concatenation of paths: $P_i(s, u) \cdot (u, v) \cdot Q(v, t)$. Note that the length of $Q(v, t)$ is at most the length of $\hat{P}_i^2(v_i, t)$. We are now ready to prove the main Theorem of this section.

THEOREM 4.1. Let $k \geq 1$ and let T be the output of the modified k -simple shortest paths algorithm. For every $i \leq k$ the length of the i -th path added to T is at most $\frac{3}{2}$ the length of $P_i(s, t)$. The running time of the algorithm is $O(k(m\sqrt{n} + n^{3/2} \log n))$.

Proof. (Sketch) Let T_{i-1} be the tree after $i - 1$ iterations of the algorithm. Let $j < i$ and let $\hat{P}_j(s, t)$ be the path in T that has the longest common prefix with $P_i(s, t)$.

Let (u_j, v_j) be the deviation edge of $\hat{P}_j(s, t)$. Notice that it is guaranteed by the modification of the algorithm, described above, that $\hat{P}_j(v_j, t)$ is the shortest path in the graph $G' = G(V \setminus \hat{P}_j(s, u_j), E \setminus E_d(v_j))$ and we can now compute an approximation for its second shortest path. Let $Q(v_j, t)$ be the approximated second shortest path computed for the path $\hat{P}_j(v_j, t)$. The path $P_i(v_j, t)$ is part of G' , thus, the length of $Q(v_j, t)$ is at most $\frac{3}{2}$ the length of $P_i(v_j, t)$. The path $\hat{P}_j(s, u_j) \cdot (u_j, v_j) \cdot Q(v_j, t)$ is added to H by the algorithm. As a result of that, the length of the minimal path extracted from H in the i -th iteration is at most $\frac{3}{2}$ of the length of $P_i(s, t)$. There are $O(k)$ executions of the approximated second shortest paths algorithm, thus, the running time is $O(k(m\sqrt{n} + n^{3/2} \log n))$.

5 Concluding remarks and open problems

We presented the first approximation algorithm for finding k -simple shortest paths in weighted directed graphs. The running time of the algorithm is $\tilde{O}(m\sqrt{n})$. The main problem, however, remains open. Is it possible to get an exact algorithm for finding the k -simple shortest paths in weighted directed graphs with a running time of $O(n^{3-\epsilon})$, for some $\epsilon > 0$? Another natural question is whether the stretch factor can be further improved. For example, is it possible to obtain an algorithm whose running time is $O(m\sqrt{n} \text{ polylog}(n, \epsilon))$ and the stretch of the paths is $1 + \epsilon$?

Acknowledgments: Many thanks to Vera Asodi and Uri Stav for reading the paper and giving many helpful remarks.

References

- [1] D. Aingworth, C. Chekuri, P. Indyk, and R. Motwani. Fast estimation of diameter and shortest paths (without matrix multiplication). *SIAM Journal on Computing*, 28:1167–1181, 1999.
- [2] A. W. Brander and Mark C. Sinclair. A comparative study of k -shortest path algorithms. In *Proc. 11th UK Performance Engineering Worksh. for Computer and Telecommunications Systems*, September 1995.
- [3] T.M. Chan. All-pairs shortest paths for unweighted undirected graphs in $o(mn)$ time. In *Proc. of 17th SODA*, pages 514–523, 2006.
- [4] E. Cohen and U. Zwick. All-pairs small-stretch paths. *Journal of Algorithms*, 38:335–353, 2001.
- [5] Ernesto de Queirós Vieira Martins, Marta Margarida Braz Pascoal, and José Luís Esteves dos Santos. Deviation algorithms for ranking shortest paths. *Int. J. Found. Comp. Sci.*, 10(3):247–263, 1999.
- [6] C. Demetrescu and M. Thorup. Oracles for distances avoiding a link-failure. In *Proc. of 13th SODA*, pages 838–843, 2002.
- [7] D. Dor, S. Halperin, and U. Zwick. All pairs almost shortest paths. *SIAM Journal on Computing*, 29:1740–1759, 2000.
- [8] M. Elkin. Computing almost shortest paths. In *Proc. of 20th PODC*, pages 53–62, 2001.
- [9] D. Eppstein. Finding the k shortest paths. *SIAM Journal on Computing*, 28(2):652–673, 1998.
- [10] E. Hadjiconstantinou and Nicos Christofides. An efficient implementation of an algorithm for finding K shortest simple paths. *Networks*, 34(2):88–101, September 1999.
- [11] J. Hershberger and S. Suri. Vickrey prices and shortest paths: what is an edge worth? In *Proc. of 42nd FOCS*, pages 252–259, 2001.
- [12] N. Katoh, T. Ibaraki, and H. Mine. An efficient algorithm for K shortest simple paths. *Networks*, 12(4):411–427, 1982.
- [13] E.L. Lawler. A procedure for computing the K best solutions to discrete optimization problems and its application to the shortest path problem. *Management Science*, 18:401–405, 1971/72.
- [14] K. Malik, A. K. Mittal, and S. K. Gupta. The k most vital arcs in the shortest path problem. *Operations Research Letters*, 8(4):223–227, 1989.
- [15] Aarni Perko. Implementation of algorithms for K shortest loopless paths. *Networks*, 16:149–160, 1986.
- [16] S. Pettie. A new approach to all-pairs shortest paths on real-weighted graphs. *Theoretical Computer Science*, 312(1):47–74, 2004.
- [17] Liam Roditty and Uri Zwick. Replacement paths and k simple shortest paths in unweighted directed graphs. In *Proc. of 32th ICALP*, pages 249–260, 2005.
- [18] Mikkel Thorup and Uri Zwick. Approximate distance oracles. *J. ACM*, 52(1):1–24, 2005.
- [19] J.Y. Yen. Finding the K shortest loopless paths in a network. *Management Science*, 17:712–716, 1970/71.
- [20] U. Zwick. A slightly improved sub-cubic algorithm for the all pairs shortest paths problem with real edge lengths. In *Proc. of 15th ISAAC*, pages 921–932, 2004.