Replacement paths and k simple shortest paths in unweighted directed graphs

Liam Roditty and Uri Zwick

School of Computer Science, Tel Aviv University, Tel Aviv 69978, Israel

Abstract. Let $G = (V, E)$ be a **directed** graph and let P be a shortest path from s to t in G. In the *replacement paths* problem we are required to find, for every edge e on P, a shortest path from s to t in G that avoids e. The only known algorithm for solving the problem, even for unweighted directed graphs, is the trivial algorithm in which each edge on the path, in its turn, is excluded from the graph and a shortest paths tree is computed from s. The running time is $O(mn + n^2 \log n)$.

The replacement paths problem is strongly motivated by two different applications:

1. The fastest algorithm to compute the k **simple** shortest paths between s and t in directed graphs [Yen71,Law72] computes the replacement paths between s and t. Its running time is $\tilde{O}(mnk)$.
2. The replacement paths problem is used to compute the *Vickrey pricing* of edges in a distributed network. It was raised as an open problem by Nisan and Ronen [NR01] whether it is possible to compute the Vickrey pricing faster than n computations of a shortest paths tree.

In this paper we present the first non-trivial algorithm for computing replacement paths in unweighted directed graphs (and in graphs with small integer weights). Our algorithm is randomized and its running time is $\tilde{O}(m\sqrt{n})$ time. This result immediately improves the running time of the two applications mentioned above in a factor of $\sqrt{n}$.

We also show how to reduce the problem of computing k simple shortest paths between s and t to $O(k)$ computations of a second simple shortest path from s to t each time in a different subgraph of G. The importance of this result is that computing a second simple shortest path may turn out to be an easier problem than computing the replacement paths, thus, we can focus our efforts to improve the k simple shortest paths algorithm in obtaining a faster algorithm for the second shortest path problem.

1 Introduction

Let $G = (V, E)$ be a graph, let $s, t \in V$ be two vertices in G, and let P be a shortest path from s to t in G. In certain scenarios, edges in the graph G may occasionally fail, and we are thus interesting in finding, for every edge e on the path P, the shortest path from s to t in G that avoids e. This problem is refereed to as the *replacement paths* problem.

The replacement paths problem for *undirected* graphs is a well studied problem. An $O(m + n \log n)$ time algorithm for the problem was given by Malik *et* *al.* [MMG89]. A similar algorithm was independently discovered, much later, by Hershberger and Suri [HS01]. Hershberger and Suri [HS01] claimed that their algorithm also works for directed graphs, but this claim turned out to be false (see Hershberger and Suri [HS02]). Nardelli *et al.* [NPW01] gave an $O(m\alpha(m,n))$ time algorithm for the problem, in a stronger model of computation, using the linear time single source shortest paths algorithm of Thorup [Tho99].

All the results mentioned above for the replacement paths problem work only for undirected graphs. This situation is partially explained by an $\Omega(m\sqrt{n})$ lower bound for the replacement paths problem for directed graphs in the *path-comparison* model of Karger *et al.* [KKP93] given by Hershberger *et al.* [HSB03]

However, in directed graphs, even when the graph is unweighted, no algorithm other than naively computing a shortest paths tree from s, excluding each time a different edge of P (i.e. the shortest path from s to t), is known. The running time of this algorithm is $O(mn + n^2 \log n)$. The replacement paths problem in directed graphs is strongly motivated by the following applications:

The fastest algorithm to compute a set of k simple shortest paths in a directed graph uses in each iteration a replacement paths algorithm. This algorithm which was given independently by Yen [Yen71] and Lawler [Law72], has a running time of $O(kn(m + n \log n))$. An $o(mn)$ algorithm for the replacement paths problem implies immediately on $o(mn)$ algorithm for the k simple shortest paths problem.

The second motivation for studying this problem is the *Vickrey pricing*. Suppose we like to find the shortest path in a directed network G from s to t, where links are owned by selfish agents. Nisan and Ronen [NR01] showed that a mechanism that offers to pay $d_{G|e=\infty}(s,t) - d_{G|e=0}(s,t)$ to the owner of link e, for any edge e on the shortest path from s to t and zero otherwise, force the link owners to reveal their true link cost. This kind of pricing is called *Vickrey pricing*. Computing the first amount for every edge in the graph with respect to s and t is equivalent to computing the replacement paths between s and t. No algorithm other then the naive algorithm is known. (For more details see Hershberger and Suri [HS01] and Demetrescu et al. [DTCR]). It was raised as an open problem by Nisan and Ronen [NR01] whether it is possible to compute the Vickrey pricing faster then the naive approach.

We present here the first non-trivial algorithm for the replacement paths problem for directed graphs. It improves immediately the running time of the two applications mentioned above. Our algorithm is randomized and its running time is $\tilde{O}(m\sqrt{n})$ time. This seemingly matches the lower bound of Hershberger *et al.* [HSB03]. Unfortunately, our algorithm works only for unweighted directed graphs, or directed graphs with small integer weights, while the lower bound of [HSB03] is for generally weighted directed graphs.

One of the ingredients used in our algorithm for the replacement paths problem is a simple sampling technique used before to develop parallel algorithms (Ullman and Yannakakis [UY91]), static algorithms (Zwick [Zwi02]) and dynamic algorithms (Henzinger and King [HK95], Baswana *et al.* [BHS02,BHS03] Roditty and Zwick [RZ04b,RZ04a]) for paths problems. This technique on its own, however, does not supply an improved algorithm for the replacement paths problems and other ideas should be used.

Demetrescu and Thorup [DT02] considered the more general problem of finding, for *every* pair of vertices $u, v \in V$ and every edge $e \in E$, a shortest path from u to v that avoids e. They devise a data structure of size $O(n^2 \log n)$ capable of answering each such query in $O(\log n)$ time. The preprocessing time needed for constructing this data structure is, however, $\tilde{O}(mn^2)$. The preprocessing time can be reduced to $\tilde{O}(mn^{1.5})$ at the price of increasing the size of the data structure to $O(n^{2.5})$. Recently, this work was slightly improved by Demetrescu et al. in [DTCR].

We also consider two variants of the replacement paths problem. Assume again that $G = (V, E)$ is a directed graph and that P is a shortest path from s to t in G. In the *restricted replacement paths* problem, we are required to find, for every edge $e = (u, v)$ on the path P, a shortest path from u to t in G that avoids e. This corresponds to a scenario in which the failure of the edge $e = (u, v)$ is only detected at u (see Figure 1(b) for example). In the *edge replacement paths* problem we are required to find, for every edge $e = (u, v)$ on the path P, a shortest path from u to v in G that avoids e, (see Figure 1(c) for example). Our $\tilde{O}(mn^{1/2})$ time algorithm for the replacement paths problem can be adapted to solve these two versions of the problem.

We next turn our attention to the k *shortest paths problem*. Given a graph $G = (V, E)$, two vertices $s, t \in V$ and an integer k, we are required to find the k shortest paths from s to t in G. Eppstein [Epp98], gave an $O(m + n \log n + k)$ time algorithm for the directed version problem. However, the paths returned by Eppstein's algorithm are not necessarily simple. i.e., they may visit certain vertices more than once. In the k *shortest simple paths problem*, the paths returned should all be simple. Katoh *et al.* [KIM82] gave an $O(k(m + n \log n))$ time algorithm for solving the k shortest simple paths problem for undirected graphs. Yen [Yen71] and Lawler [Law72] gave an $O(kn(m + n \log n))$ time algorithms for solving the problem for directed graphs. It is interesting to note that, as for the replacement paths problem, the directed version here seems to be much harder. Using our $\tilde{O}(m\sqrt{n})$ time algorithm for the replacement paths problem we obtain a randomized $\tilde{O}(km\sqrt{n})$ time algorithm for the k shortest simple paths problem for unweighted directed graphs and for directed graphs with small integer weights.

Another interesting result about finding the k shortest paths from s to t in G is that computing the replacement paths is actually unnecessary. We show that computing the k simple shortest paths can be reduced to $O(k)$ computations of a second simple shortest path between s and t each time in a different subgraph of G. This implies that we can focus our efforts to obtain an $o(mn)$ algorithm for the second simple shortest path problem.

The rest of this paper is organized as follows. In the next section we describe our replacement paths algorithm and its adaption to the different variations of the replacement paths problem mentioned above. In Section 3 we explain how to compute k simple shortest paths between two vertices in $\tilde{O}(km\sqrt{n})$ and how to

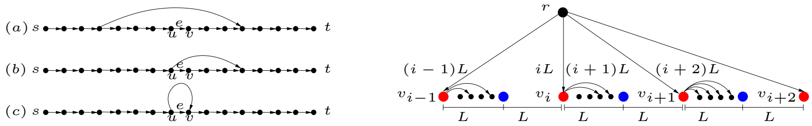


Fig. 1. Three detours and the auxiliary graph used to find short detours.

compute it only by $2k$ second shortest paths computations. We end in Section 4 with some concluding remarks and open problems.

2 Replacements paths

In this section we describe an algorithm for solving the replacement paths problem for unweighted directed graphs and directed graphs with small integer weights. Let $G = (V, E)$ be a directed graph and let s and t be two vertices in the graph. Let $P(s, t) = \langle u_0, u_1, \ldots, u_\ell \rangle$ be a shortest path from $s = u_0$ to $t = u_\ell$. Let $P^E(s, t) = \langle (u_0, u_1), (u_1, u_2), \ldots, (u_{l-1}, u_\ell) \rangle$ be the set of edges of this path. The objective of a replacement path algorithm is to find for every edge $e \in P^E(s, t)$ a shortest path $P'(s, t)$ in the graph $G_e = (V, E \setminus \{e\})$.

We start by defining a detour:

Definition 1 (A Detour). *Let $P(s, t)$ be a simple path from s to t. A simple path $D(u, v)$ is a detour of $P(s, t)$ if $D(u, v) \cap P(s, t) = \{u, v\}$ and u precedes v on $P(s, t)$.*

In Figure 1 there are three examples of possible detours of a shortest path from s to t. It is easy to see that there is a shortest path from s to t that avoids the edge (u_i, u_{i+1}) of the shortest path $P(s, t) = \langle u_0, u_1, \ldots, u_\ell \rangle$. This path is composed of an initial portion $\langle u_0, u_1, \ldots, u_j \rangle$ of P, where $0 \le j \le i$, a detour $D(u_j, u_{j'})$, where $i + 1 \le j' \le \ell$, and then the final portion $\langle u_{j'}, \ldots, u_\ell \rangle$ of P. Our goal is to find a detour that minimizes the length of P.

Let L be a parameter to be chosen later. A detour is said to be *short* if its length is at most L. Otherwise, it is said to be *long*. (Note that we are considering here only the length of the detour, not the total length of the resulting path from s to t. For example, the detour in Figure 1(a) is longer than the one in Figure 1(b), however, the resulting paths may have the same length.)

We find separately the best short detours and the best long detours. The short detours are found in Section 2.1 in $\tilde{O}(mL)$ time. The long ones are found in Section 2.2 in $\tilde{O}(mn/L)$ time. Setting $L = \sqrt{n}$, we get that the running time of both algorithms is $\tilde{O}(m\sqrt{n})$. By choosing for every edge the best short or long detour, we obtain all the optimal replacement paths.

2.1 Finding short detours

We now describe an $\tilde{O}(mL)$ time algorithm for finding the best detours of length at most L. We can easily find the best detours that start in a given vertex u on the shortest path $P(s, t)$ by running the BFS algorithm from u in the graph $G - P^E$. However, doing so from each vertex on P may require n BFS computation which is too time consuming. The main observation made in this

algorithm **ShortDet**(P, b, L)	algorithm **ShortRepPath**(P, L)

$\langle u_0, u_1, \ldots, u_\ell \rangle \leftarrow P$
$V' \leftarrow V \cup \{r\}$
$E' \leftarrow E \setminus P^E$
for each $e \in E'$ do $wt(e) \leftarrow 1$
for $i \leftarrow 0$ to $\lfloor (\ell - b)/(2L) \rfloor$
  $E' \leftarrow E' \cup \{(r, u_{2iL+b})\}$
  $wt(r, u_{2iL+b}) \leftarrow iL$
$\delta' \leftarrow$ **Dijkstra**$(r, (V', E', wt))$
for $i \leftarrow 0$ to $\lfloor (\ell - b)/(2L) \rfloor$
  $g \leftarrow 2iL + b$
  for $j \leftarrow 1$ to L
    if $\delta'(r, u_{g+j}) \leq (i+1)L$ then
      $RD[g, j] \leftarrow \delta'(r, u_{g+j}) - iL$
    else
      $RD[g, j] \leftarrow \infty$
	$\langle u_0, u_1, \ldots, u_\ell \rangle \leftarrow P$
	for $b \leftarrow 0$ to $2L - 1$
	  ShortDet(P, b, L)
	$Q \leftarrow \phi$
	for $i \leftarrow 0$ to $\ell - 1$
	  for $j \leftarrow i + 1$ to $\min\{i + L, \ell\}$
	    $Insert(Q, (i, j), RD[i, j - i] + i + \ell - j)$
	  for $j \leftarrow \max\{i - L, 0\}$ to $i - 1$
	    $Delete(Q, (j, i))$
	  $(a, b) \leftarrow findmin(Q)$
	  $len \leftarrow RD[a, b - a] + a + \ell - b$
	  $RP[i] \leftarrow \langle len, a, b \rangle$

Fig. 2. The algorithm for finding short detours and short replacement paths

Section is that if $v_0, v_1, \ldots, v_k$ are vertices on $P(s,t)$ that are at a distance of at least $2L$ apart from each other, then the best short detours from all these vertices can be found by one run of Dijkstra's algorithm on a suitably modified graph. Thus, $O(L)$ runs suffice to find all short detours.

More specifically, to find the best detours from the vertices $u_0, u_{2L}, \ldots, u_{2kL}$, were $k = \lfloor \frac{\ell}{2L} \rfloor$, we consider the graph $G - P^E$ to which we add a new source vertex r and an edge (r, u_{2iL}) of weight iL, for every $0 \leq i \leq k$. The weight of all the edges of $E - P^E$ is set to 1. Note that the weight assigned to the edge (r, u_{2iL}) is iL, and not $2iL$ as might have been expected. Also, note that even though we are interested in detours that are of length at most L, the distance between every two selected vertices should be at least $2L$. The reason to that will become clear in the proof of Theorem 1. The resulting auxiliary graph, which we denote by G^A, is depicted in Figure 1.

We claim that by running Dijkstra's algorithm, from r, on G^A we find all the best short detours that start in one of the selected vertices. We then run this algorithm $2L-1$ more times to find short detours emanating from the other vertices of $P(s,t)$. In the i-th run we find the short detours emanating from one of the vertices $u_i, u_{2L+i}, \ldots, u_{2kL+i}$.

We let $\delta(u,v)$ denote the distance from u to v in the graph G. We let $\delta^-(u,v)$ denote the distance from u to v in the graph $G - P^E$, i.e., the graph G with the edges of the path P removed. (The minus sign is supposed to remind us that the edges of P are removed from the graph.) We let $\delta^A(u,v)$ denote the distance from u to v in the auxiliary graph G^A. We now claim:

Theorem 1. *If* $\delta^A(r, u_{2iL+j}) \leq (i+1)L$, *where* $0 \leq i \leq k$ *and* $1 \leq j \leq L$, *then* $\delta^-(u_{2iL}, u_{2iL+j}) = \delta^A(r, u_{2iL+j}) - iL$. *Otherwise,* $\delta^-(u_{2iL}, u_{2iL+j}) > L$.

Proof. For brevity, let $v_i = u_{2iL}$ and $v_{ij} = u_{2iL+j}$. Assume at first that $\delta^A(r, v_{ij}) \leq (i+1)L$. Consider a shortest path from r to v_{ij} in G^A. Let (r, v_q) be the first edge on the path. If $q < i$, then we have

$$\delta^A(r, v_{ij}) = qL+\delta^-(v_q, v_{ij}) \geq qL+2(i-q)L+j = (2i-q)L+j \geq (i+1)L+j > (i+1)L \ ,$$

a contradiction. Note that if the distance between any v_i and v_{i+1} was L instead of $2L$ then for $q < i$ we do have $\delta^A(r, v_{ij}) \leq (i+1)L$.

Similarly, if $q > i$ then we again have

$$\delta^A(r, v_{ij}) = qL + \delta^-(v_q, v_{ij}) > (i+1)L \ .$$

Thus, we must have $q = i$ and $\delta^A(r, v_{ij}) = iL + \delta^-(v_i, v_{ij})$, as required.

On the other hand, if $\delta^-(v_i, v_{ij}) \leq L$, then clearly

$$\delta^A(r, v_{ij}) \ \leq \ wt(r, v_i) + \delta^-(v_i, v_{ij}) \ \leq \ iL + L \ = \ (i+1)L \ ,$$

as required.

A description of the resulting algorithm, which we call **ShortDet** is given in Figure 2. By running the algorithm with the parameter b ranging from 0 to $2L-1$ we find, for every vertex on the path, the best short detours starting at it. This information is gathered in the table RD.

The entry $RD[i, j]$ gives us the length of the shortest detour starting at u_i and ending at u_{i+j}, if that length is at most L. To find the shortest path from s to t that avoids the edge (u_i, u_{i+1}) and uses a short detour, we need to find indices $i - L \leq a \leq i$ and $i < b \leq i + L$ for which the expression

$$\delta(s, u_a) + \delta^-(u_a, u_b) + \delta(u_b, t) \ = \ a + RD[a, b-a] + (\ell - b)$$

is minimized. An algorithm, called **ShortRepPath**, for finding such replacement paths is given in Figure 2. Algorithm **ShortRepPath** uses a priority queue Q. When looking for the shortest replacement path for the edge (u_i, u_{i+1}), the priority queue Q contains all pairs (a, b) such that $i-L \leq a \leq i$ and $i < b \leq i+L$. The key associated with a pair (a, b) is naturally $a+RD[a, b-a]+(\ell-b)$. In the start of the iteration corresponding to the edge (u_i, u_{i+1}), we insert the pairs (i, j), for $i+1 \leq j \leq i+L$ into Q, and remove from it the pairs (j, i), for $i-L \leq j \leq i$. A $findmin$ operation on Q then returns the minimal pair (a, b). It is easy to see that the complexity of this process is only $\tilde{O}(nL)$. Thus, the total running time of the algorithm is $\tilde{O}(mL)$, as required. We have thus proved:

Theorem 2. *Algorithm* **ShortRepPath** *finds all the shortest replacement paths that use short detours. Its running time is* $\tilde{O}(mL)$.

2.2 Finding long detours

To find long detours, i.e., detours that are of length at least L, we use the following simple sampling lemma. (To the best of our knowledge, it was not used before in the context of finding replacement paths).

Lemma 1. *Let $D_1, D_2, \ldots, D_q \subseteq V$ such that $|D_i| \geq L$ for $1 \leq i \leq q$ and $|V| = n$. If $R \subseteq V$ is a random subset obtained by selecting each vertex, independently, with probability $(c \ln n)/L$, for some constant c, then with probability of at least $1 - q \cdot n^{-c}$ we have $D_i \cap R \neq \phi$ for every $1 \leq i \leq q$.*

For every pair of vertices u and v on the path P for which the shortest detour from u to v is of length at least L, let $D(u, v)$ be such a shortest detour. By the lemma, if R is a random set as above, then with a probability of at least $1 - n^{2-c}$ we have $D(u, v) \cap R \neq \phi$, for every such pair u and v. The choice of the random set R is the only randomization used by our algorithm.

Our algorithm for finding the best replacement paths that use long detours is given in Figure 4 in the Appendix. The algorithm starts by calling *sample*$(V, (4 \ln n)/L)$ which selects a random set R in which each vertex $v \in V$ is placed, independently, with probability $(4 \ln n)/L$. The expected size of R is clearly $\tilde{O}(\frac{n}{L})$. We assume, throughout the section, that $D(u, v) \cap R \neq \phi$, whenever $|D(u, v)| \geq L$.

For every sampled vertex $r \in R$, the algorithm maintains two priority queues $Q_{in}[r]$ and $Q_{out}[r]$ containing indices of vertices on P. When looking for a replacement path for the edge (u_i, u_{i+1}) we have $Q_{in}[r] = \{0, 1, \ldots, i\}$ and $Q_{out}[r] = \{i + 1, \ldots, \ell\}$. The key associated with an element $j \in Q_{in}[r]$ is $j + \delta^-(u_j, r)$. The key associated with an element $j \in Q_{out}[r]$ is $\delta^-(r, u_j) + (\ell - j)$.

Recall that $\delta^-(u, v)$ is the distance from u to v in $G - P^E$. The algorithm computes $\delta^-(r, v)$ and $\delta^-(v, r)$, for every $r \in R$ and $v \in V$, by running two BFS's from r, for each $r \in R$, one in $G - P^E$ and one in the graph obtained from $G - P^E$ by reversing all the edges. (Only one of these BFS's is explicitly mentioned in **LongRepPath**.) The total running time of computing these distances is $\tilde{O}(mn/L)$.

To find the shortest replacement path for the edge (u_i, u_{i+1}) that passes through a given vertex $r \in R$, the algorithm needs to find an index $0 \leq a \leq i$ which minimizes the expression $a + \delta^-(u_a, r)$, and an index $i < b \leq \ell$ which minimizes the expression $\delta^-(r, u_b) + (\ell - b)$. The minimizing index a is found by a $findmin$ operation on $Q_{in}[r]$ and the minimizing index b is found by a $findmin$ operation on $Q_{out}[r]$.

It is not difficult to check that the total running time of the algorithm is $\tilde{O}(mn/L)$, as required. We have thus proved:

Theorem 3. *Algorithm* **LongRepPath** *finds, with very high probability, all the shortest replacement paths that use long detours. Its running time is $\tilde{O}(mn/L)$.*

2.3 The replacement paths algorithm and its variants

The algorithms **ShortRepPath** and **LongRepPath** find the best short and long replacement paths available to bypass every edge on a given shortest path. By passing on their output and picking the minimal path found for every edge we obtain the solution for the replacement paths problem as promised.

There are another two natural variants of replacement paths that can be solved by our short and long detours detection.

Let $G = (V, E)$ be a directed graph and let P be a shortest path from s to t in G. In the *restricted replacement paths* problem, we are required to find, for every edge $e = (u, v)$ on the path P, a shortest path from u to t in G that avoids e. This corresponds to a scenario in which the failure of the edge $e = (u, v)$ is only detected at u. In the *edge replacement paths* problem we are required to find, for every edge $e = (u, v)$ on the path P, a shortest path from u to v in G that avoids e.

To solve the above two problems the main idea of short and long detours remains unchanged. The only change is in the set from which we choose the best detour to be used to bypass a given edge. This set is now updated according to the structural restrictions given by the problem definition.

For the restricted replacement paths after finding the short detours by the algorithm **ShortDet** we maintain the heap subject to the following constraint: If we currently searching for a restricted replacement path to bypass the edge (u_i, u_{i+1}) the heap contains only detours that emanate from u_i.

In a similar manner, when searching for a restricted replacement path composed from a long detour to bypass the edge (u_i, u_{i+1}) we use only detours that emanate from u_i. Thus, we only need one heap for this process. The algorithm is given in Figure 5 in the Appendix. We claim:

Theorem 4. *Algorithm* **ResRepPath***, with $L = \sqrt{n}$, finds, with very high probability, all the restricted shortest replacement paths. Its running time is $\tilde{O}(m\sqrt{n})$.*

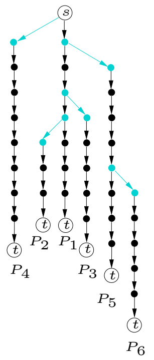
Using similar constraints on the set from which paths are picked we can adapt our ideas to solve also the edge replacement paths problem.

3 The k shortest simple paths problem

The k shortest simple paths problem (also known as the k shortest loopless paths) is a fundamental graph theoretic problem. Let $G = (V, E)$ be a directed graph and let s and t be two vertices of the graph. Let k be an integer. The target is to find the k shortest simple paths from s to t. This version of the problem is considered to be much harder than the general version in which non-simple paths (i.e. paths that may contain a loop) are allowed to be among the k shortest paths. The k shortest non-simple paths can be computed in time of $O(m+n\log n+k)$ using an algorithm of Eppstein [Epp98]. In cases that a shortest paths tree can be computed in $O(m + n)$, Eppstein's algorithm has a running time of $O(m + n + k)$. However, the running time of the restricted problem is much worse. The best algorithm is due to Yen [Yen71] and Lawler [Law72]. It has a running time of $O(kn(m + n \log n))$.

In this section we show that for unweighted directed graphs (and for graphs with small integer weights) the running time of $O(kn(m + n \log n))$ can be significantly improved using our new replacement paths algorithm. We obtain a randomized algorithm with running time of $O(km\sqrt{n} \log n)$.

We also reduce the problem of computing k simple shortest paths to $O(k)$ computations of a second shortest path each time in a different subgraph of G. This reduction works in weighted graphs. Both Yen [Yen71] and Lawler [Law72]



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

$P_1(s,t) \leftarrow$ **Dijkstra**$(G(V,E),s,t)$
$T \leftarrow P_1(s,t)$
$Insert(Q, (\mathbf{SP}(G(V,E), P_1(s,t)), 1))$
for $i \leftarrow 2$ to k
  $(P_i(s,t), j) \leftarrow findmin(Q)$
  Make a copy of $P_i(v_i,t)$ and hang it on u_i in T
  $Insert(Q, (\mathbf{SP}(G(V, E \setminus E_d(v_i)), P_i(v_i,t)), i))$
  $Insert(Q, (\mathbf{SP}(G(V, E \setminus E_d(v_j)), P_j(v_j,t)), j))$

Fig. 3. A deviations tree and our k-simple shortest paths algorithm

use $O(k)$ computations of replacement paths. Our reduction implies that we can focus our efforts in improving the second shortest path algorithm, which may turn out to be an easier problem than the replacement paths problem. We only deal in this section with simple paths thus we refer to a simple path simply by saying a path.

The algorithm for computing k shortest paths works as follow. It maintains a priority queue Q of paths from s to t. The key attached with each path is its length. The algorithm preforms k iterations. The priority queue is initialized with a second shortest path of a shortest path from s to t. In the i-th iteration the algorithm picks from Q the path with the minimal length and remove it. Let $P_i(s,t) = \langle s, u_1, \ldots u_{\ell-1}, t\rangle$ be the i-th path picked by the algorithm. This path is added to the output as the i-th shortest path. To describe the output structure we need the following definition:

Definition 2 (Deviation edge and Deviations tree). *For $k=1$ the deviations tree is simply a copy of a shortest path from s to t. Suppose that the tree already exists for $i-1$ paths. Let $P_i(s,t) = \langle s, u_1, \ldots, u_l\rangle$ be the i-th shortest path to be output. Let $P_i(s,u_j)$ be the longest subpath of $P_i(s,t)$ that was already part of the output and thus part of the tree. We make a copy only from $P_i(u_{j+1},t)$ and hang it on the copy of u_j in the tree. We say that the edge (u_j, u_{j+1}) is the deviation edge of $P_i(s,t)$.*

Note that by this definition a vertex may have more than one occurrence in the deviations tree. However, there are at most k copies of each vertex in the deviations tree, thus, the size of the tree is $O(kn)$. An example of a deviations tree is given in Figure 3. The deviation edges are in light color.

The main challenge is to obtain fast the paths to be added to Q in each iteration. Suppose we have extracted the i-th shortest path $P_i(s,t)$ from Q. After having $P_i(s,t)$ in the deviations tree we need to find the new paths to be added to Q. Let (u_i, v_i) be the deviation edge of $P_i(s,t)$. In Yen's algorithm the path $P_i(u_i,t) = \langle w_1, w_2, \ldots, w_l\rangle$, where $w_1 = u_i$ and $w_l = t$, is scanned. For each vertex $w_j \in P_i(u_i,t)$, the algorithm finds a shortest path $P'(w_j,t)$ from w_j to t which does not use the edge (w_j, w_{j+1}). In the special case of w_1 the path $P'(w_1,t)$ is obtained when all the edges emanate from the copy of w_1 in the tree are forbidden to use. Each such a path is concatenated to $P_i(s, w_j)$ and added to Q. This is essentially a restricted replacement paths problem for the path $P_i(u_i, t)$. Thus, we can claim the following:

Theorem 5. *The algorithm of Yen combined with our restricted shortest paths algorithm computes k simple shortest paths in $\tilde{O}(km\sqrt{n})$ time.*

However, this process generates many paths and most of them are not needed. Many of these paths can be ruled out without being actually computed by making just two computations of second shortest paths in the i-th iteration.

Recall that $P_i(s, t)$ is the i-th shortest path extracted from Q and (u_i, v_i) is its deviation edge. Let $E_d(v_i)$ be the set of deviation edges emanate from $P_i(v_i, t)$. We compute a second shortest path for the path $P_i(v_i, t)$ in the graph $G(V, E \setminus E_d(v_i))$, concatenate it to $P_i(s, v_i)$ and add it to Q. This path is associated to $P_i(s, t)$. Note that in the first computation of a second shortest path for $P_i(v_i, t)$ the set $E_d(v_i)$ is essentially empty. However, we are not done yet. The extracted path $P_i(s, t)$ is associated to some other path $P_j(s, t)$, where $j < i$. Since we have extracted the path associated to $P_j(s, t)$ we need to find a new path, other than $P_i(s, t)$, to associate to $P_j(s, t)$. Let (u_j, v_j) be the deviation edge of $P_j(s, t)$. We compute a second shortest path for the path $P_j(v_j, t)$ in the graph $G(V, E \setminus E_d(v_j))$. By its definition the set $E_d(v_j)$ contains in this stage the deviation edge of $P_i(s, t)$, thus, the resulting second shortest path will be other than $P_i(s, t)$. We concatenate this path to $P_j(s, v_j)$ and add it to Q. The algorithm is given in Figure 3. We assume that the deviation edge of the path $P_i(s, t)$ if exists is (u_i, v_i). For the path $P_1(s, t)$ we treat s as the head of a deviation edge. We use the algorithm of $Dijkstra$ to compute a shortest path from s to t and the algorithm SP to compute a second shortest path given a shortest path.

Next, we justify why the extracted path is only associated to one other path or more precisely why a path cannot be added to Q as a second shortest path of two different paths.

Lemma 2. *In any stage all paths in Q are distinct.*

Proof. To the purpose of the proof only we divide the paths of the graph into disjoint sets. Each set is associated with a path which already was picked from Q. The set C_i is associated with the path $P_i(s, t)$. We prove that these sets exist and any second shortest path in Q is associated to a different set. This implies that all paths in Q are distinct.

We set C_1 to be all the paths in the graph. After finding a second shortest path for the first time we divide C_1 into two sets. Let (u_2, v_2) be the deviation edge of $P_2(s, t)$ then C_2 is set to be all the paths from C_1 that have the prefix $P_2(s, v_2)$ and C_1 is set to $C_1 \setminus C_2$. Obviously, C_1 and C_2 are disjoint. Now computing a second shortest path for $P_2(v_2, t)$ and concatenating it with $P_2(s, v_2)$ results in a path from C_2 and computing a second shortest path for the path $P_1(s, t)$ in $G(V, E \setminus E_d(s))$ results in a second shortest path in C_1. (Note that $E_d(s)$ contains the edge (u_2, v_2).) Thus, the paths added to Q are from two disjoint sets. We prove by induction that in general it also holds.

Suppose that right before the i-th extraction we have $i-1$ disjoint sets, such that for any $j \leq i-1$ the set C_j is associated to $P_j(s,t)$ and a path composed from the concatenation of $P_j(s,v_j)$ and a second shortest path of $P_j(v_j,t)$ in the graph $G(V, E \setminus E_d(v_j))$ is the path in Q from C_j. We show that right before the $i+1$-th extraction this invariant still holds. Let $P_i(s,t)$ be the path picked from Q in the i-th extraction. By the induction hypothesis we know that there is a set C_j, disjoint from all the others, such that $P_i(s,t) \in C_j$ and $P_j(s,t)$ is associated with C_j. Let (u_i, v_i) be the deviation edge of $P_i(s,t)$. We divide C_j into two sets as follow, C_i will have the paths from C_j with the prefix $P_i(s,v_i)$ and C_j is set to $C_j \setminus C_i$. The resulted sets are disjoint. Now computing a second shortest path for $P_i(v_i,t)$ and concatenating it with $P_i(s,v_i)$ results in a path from C_i and computing a second shortest path for the path $P_j(v_j,t)$ in $G(V, E \setminus E_d(v_j))$ and concatenating it with $P_j(s,v_j)$ results in a second shortest path in C_j. In this process the two paths added to Q are from different disjoint sets and the above invariant still holds.

It follows that once a path is out of Q only two second shortest path computations have to be done. We now claim the correctness of the algorithm.

Lemma 3. *The algorithm computes k simple shortest paths.*

Proof. The proof is by induction. For $i = 1$ the claim trivially holds. Suppose that the $i-1$ first paths are found by our algorithm. We prove that the i-th path is found also. Let W be the weight of the i-th shortest path. Let $P_i(s,t)$ be an i-th shortest path. Let (u_i, v_i) be its deviation edge and let $P_j(s,t)$ be the path $P_i(s,t)$ deviates from. We will show that the path $P_i(s,t)$ or other path of the same length must be associated to $P_j(s,t)$.

Let (u', v') be the closest deviation edge to (u_i, v_i) on the path $P_j(s, u_i)$, if exists, or let $v' = s$ otherwise. Consider the last time a second shortest path computation was done for the path $P_j(v', t)$ before the i-th extraction. The weight of the path P obtained in this computation is at most W since all the edges of $P_i(u_i, t)$ are eligible to use. Suppose that the length of P is strictly less than W then by the induction hypothesis the path P is extracted before the i-th extraction. By our algorithm when P is extracted we recompute a second shortest path for $P_j(v', t)$, the path that P was associated to. However, the second shortest path computation that have added P was the last computation done for $P_j(v', t)$ before the i-th extraction, a contradiction.

The following Theorem stems from Lemma 2 and Lemma 3.

Theorem 6. *The algorithm described above computes correctly the k simple shortest paths of a directed graph by $O(k)$ computation of second shortest paths.*

4 Concluding remarks and open problems

We presented a randomized $O(m\sqrt{n})$ times algorithm for the replacement paths problem in unweighted graphs and in graphs with small integer weights. However, many problems still remain open, among them:

1. Is it possible to obtain similar results for weighted graphs?
2. Is it possible to compute a second shortest path in a weighted graph in $o(mn)$? This will imply an $o(mn)$ time algorithm for finding a set of k simple shortest paths.

References

[BHS02] S. Baswana, R. Hariharan, and S. Sen. Improved decremental algorithms for transitive closure and all-pairs shortest paths. In *Proc. of 34th STOC*, pages 117–123, 2002.

[BHS03] S. Baswana, R. Hariharan, and S. Sen. Maintaining all-pairs approximate shortest paths under deletion of edges. In *Proc. of 14th SODA*, pages 394–403, 2003.

[DT02] C. Demetrescu and M. Thorup. Oracles for distances avoiding a link-failure. In *Proc. of 13th SODA*, pages 838–843, 2002.

[DTCR] C. Demetrescu, M. Thorup, R. Alam Chaudhury, and V. Ramachandran. Oracles for distances avoiding a link-failure.

[Epp98] D. Eppstein. Finding the k shortest paths. *SIAM Journal on Computing*, 28(2):652–673, 1998.

[HK95] M. Henzinger and V. King. Fully dynamic biconnectivity and transitive closure. In *Proc. of 36th FOCS*, pages 664–672, 1995.

[HS01] J. Hershberger and S. Suri. Vickrey prices and shortest paths: what is an edge worth? In *Proc. of 42nd FOCS*, pages 252–259, 2001.

[HS02] J. Hershberger and S. Suri. Erratum to "vickrey pricing and shortest paths: What is an edge worth?". In *Proc. of 43rd FOCS*, page 809, 2002.

[HSB03] J. Hershberger, S. Suri, and A. Bhosle. On the difficulty of some shortest path problems. In *Proc. of the 20th STACS*, pages 343–354, 2003.

[KIM82] N. Katoh, T. Ibaraki, and H. Mine. An efficient algorithm for K shortest simple paths. *Networks*, 12(4):411–427, 1982.

[KKP93] D.R. Karger, D. Koller, and S.J. Phillips. Finding the hidden path: time bounds for all-pairs shortest paths. *SIAM Journal on Computing*, 22:1199–1217, 1993.

[Law72] 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.

[MMG89] 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.

[NPW01] E. Nardelli, G. Proietti, and P. Widmayer. A faster computation of the most vital edge of a shortest path. *Information Processing Letters*, 79(2):81–85, 2001.

[NR01] N. Nisan and A. Ronen. Algorithmic mechanism design. *Games and Economic Behavior*, 35:166–196, 2001.

[RZ04a] L. Roditty and U. Zwick. Dynamic approximate all-pairs shortest paths in undirected graphs. In *Proc. of 45th FOCS*, 2004. 499-508.

[RZ04b] L. Roditty and U. Zwick. On dynamic shortest paths problems. In *Proc. of 12th ESA*, 2004. 580–591.

[Tho99] M. Thorup. Undirected single-source shortest paths with positive integer weights in linear time. *Journal of the ACM*, 46:362–394, 1999.

[UY91] J.D. Ullman and M. Yannakakis. High-probability parallel transitive-closure algorithms. *SIAM Journal on Computing*, 20:100–125, 1991.

[Yen71] J.Y. Yen. Finding the K shortest loopless paths in a network. *Management Science*, 17:712–716, 1970/71.

[Zwi02] U. Zwick. All-pairs shortest paths using bridging sets and rectangular matrix multiplication. *Journal of the ACM*, 49:289–317, 2002.

```
algorithm LongRepPath(P, L)

⟨u_0, u_1, ..., u_ℓ⟩ ← P
R ← sample(V, (4 ln n)/L)
for each r ∈ R do
  δ⁻ ← BFS(r, G − P^E)
  Q_in[r] ← φ
  Q_out[r] ← φ
  for j ← 0 to ℓ
    Insert(Q_out[r], j, δ⁻(r, u_j) + (ℓ − j))
for i ← 0 to ℓ − 1
  len ← ∞
  for each r ∈ R do
    Insert(Q_in[r], i, i + δ⁻(u_i, r))
    Delete(Q_out[r], i)
    a ← findmin(Q_in[r])
    b ← findmin(Q_out[r])
    len' ← a + δ⁻(u_a, r) + δ⁻(r, u_b) + (ℓ − b)
    if len < len' then
      ⟨len, st, mid, end⟩ ← ⟨len', a, mid, b⟩
  RP[i] ← ⟨len, st, mid, end⟩
```

Fig. 4. Finding long detours and long replacment paths

```
algorithm ResRepPath(P, L)

⟨u_0, u_1, ..., u_ℓ⟩ ← P
for b ← 0 to 2L − 1
    ShortDet(P, b, L)
R ← sample(V, (4 ln n)/L)
for each r ∈ R do
    δ⁻ ← BFS(r, G − P^E)
    Q[r] ← φ
    for j ← 0 to ℓ
        Insert(Q[r], j, δ⁻(r, u_j) + (ℓ − j))
for i ← 0 to ℓ − 1
    len ← ∞
    for each r ∈ R do
        Delete(Q[r], i)
        a ← findmin(Q[r])
        len' ← δ⁻(u_i, r) + δ⁻(r, u_a) + (ℓ − a)
        if len' < len then
            ⟨len, st, mid, end⟩ ← ⟨len, i, mid, a⟩
    for j ← i + 1 to L + i + 1
        if RD(i, j − i) + (ℓ − j) < len then
            ⟨min, st, end⟩ ← ⟨RD(i, j − i) + (ℓ − j), i, j⟩
    DP[u_i] ← ⟨len, st, end⟩
```

Fig. 5. Finding shortest restricted replacement paths