Near Optimal Algorithm for the Directed Single Source Replacement Paths Problem

Shiri Chechik
Blavatnik School of Computer Science, Tel Aviv University, Israel
shiri.chechik@gmail.com

Ofer Magen
Blavatnik School of Computer Science, Tel Aviv University, Israel
ofermagen98@gmail.com

Abstract

In the Single Source Replacement Paths (SSRP) problem we are given a graph $G = (V, E)$, and a shortest paths tree $\widehat{K}$ rooted at a node s, and the goal is to output for every node $t \in V$ and for every edge e in $\widehat{K}$ the length of the shortest path from s to t avoiding e.

We present an $\tilde{O}(m\sqrt{n} + n^2)$ time randomized combinatorial algorithm for unweighted directed graphs[1]. Previously such a bound was known in the directed case only for the seemingly easier problem of replacement path where both the source and the target nodes are fixed.

Our new upper bound for this problem matches the existing conditional combinatorial lower bounds. Hence, (assuming these conditional lower bounds) our result is essentially optimal and completes the picture of the SSRP problem in the combinatorial setting.

Our algorithm naturally extends to the case of small, rational edge weights. In the full version of the paper, we strengthen the existing conditional lower bounds in this case by showing that any $O(mn^{1/2-\epsilon})$ time (combinatorial or algebraic) algorithm for some fixed $\epsilon > 0$ yields a truly sub-cubic algorithm for the weighted All Pairs Shortest Paths problem (previously such a bound was known only for the combinatorial setting).

2012 ACM Subject Classification Theory of computation → Dynamic graph algorithms

Keywords and phrases Fault tolerance, Replacement Paths, Combinatorial algorithms, Conditional lower bounds

Digital Object Identifier 10.4230/LIPIcs.ICALP.2020.81

Category Track A: Algorithms, Complexity and Games

Related Version A full version of the paper is available at `https://arxiv.org/abs/2004.13673`.

Funding This publication is part of a project that has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No 803118 UncertainENV).

1 Introduction

In the replacement paths (RP) problem, we are given a graph G and a shortest path P between two vertices s and t and the goal is to return for every edge e in P the length $d(s, t, G - e)$, where $G - e$ is the graph obtained by removing the edge e from G, and $d(s, t, G - e)$ is the distance between s and t in the resulted graph. In some cases the goal is to provide the shortest path itself and not only its length. The interest in replacement path problems stems from the fact that failures and changes in real world networks are inevitable, and in many cases we would like to have a solution or a data structure that can

[1] As usual, n is the number of vertices, m is the number of edges and the $\tilde{O}$ notation suppresses poly-logarithmic factors.

adapt to these failures. The replacement paths problem is a notable example where we would like to have backup paths between two distinguished vertices in the event of edge failures. The replacement paths problem is also very well motivated as it is used as an important ingredient in other applications such as the Vickrey pricing of edges owned by selfish agents from auction theory [14, 6]. Another application of the replacement path problem is finding the k shortest simple paths between a pair of vertices. The k shortest simple paths problem can be solved by invoking the replacement paths algorithm k times and adding a very small weight to the path found in each invocation. The k shortest simple paths problem has many applications by itself [4]. The replacement paths problem has been extensively studied and the literature covers many aspects of this problem with many near optimal solutions in many of the cases (see e.g. [11, 13, 12, 15, 3, 9, 19, 16, 2]).

In this paper we consider a natural and important generalization of the replacement paths problem, referred to as the single source replacement paths (SSRP) problem, which is defined as follows. Given a graph G and a shortest paths tree $\widehat{K}$ rooted at a node s, the SSRP problem is to compute the values of $d(s,t,G-e)$ for every vertex $t \in V(G)$ and for every edge $e \in E(\widehat{K})$. Note that as the number of edges in $\widehat{K}$ is $n-1$, there are $O(n^2)$ different distances we need to evaluate. It follows that the size of the SSRP output is $O(n^2)$.

Despite of its natural flavor, the picture of the SSRP problem is not yet complete in many of the cases. To the best of our knowledge the first paper that considered the SSRP problem is by Hershberger et al. [7] who referred to the problem as edge-replacement shortest paths trees and showed that in the path-comparison model of computation of Karger et al. [8], there is a lower bound of $\Omega(mn)$ comparisons in order to solve the SSRP problem for arbitrarily weighted directed graphs.

For the directed weighted case it was shown by Vassilevska Williams and Williams [18] that any truly sub-cubic algorithm for the simpler problem of RP in **directed**, arbitrarily weighted graph admits a truly sub-cubic algorithm for the arbitrarily weighted All Pairs Shortest Paths (APSP) problem. The conditional lower bound from [18] holds only for the directed case, and quite interestingly for the undirected arbitrarily weighted case, the classical RP problem admits a near linear time algorithm [11, 13, 12]. However, the SSRP problem in undirected graphs appears to be much harder than the RP problem. In [2] it was shown by Chechik and Cohen that any truly sub-cubic solution for the SSRP problem in **undirected** arbitrarily weighted graphs, admits a truly sub-cubic algorithm for the arbitrarily weighted APSP problem. Therefore, it seems there is no hope to solve the SSRP problem in weighted graphs, both in the directed and undirected case. Meaning that if we seek for truly sub-cubic algorithms for the SSRP problem we must either consider unweighted graphs or restrict the edge weights in some other way.

One way to restrict the weights is to consider only bounded **integer** edge weights. This restriction was considered by Grandoni and Vassilevska Williams [5], who were also the ones to name this problem the single source replacement paths problem. Grandoni and Vassilevska Williams [5] gave the first non trivial upper bound for the SSRP problem. They showed that one can bypass the cubic lower bounds by using fast matrix multiplications and by restricting the weights to be integers in a bounded range. More precisely, they showed that for graphs with positive integer edge weights in the range $[1,M]$, SSRP can be computed in $\tilde{O}(Mn^{\omega})$ time (here $\omega < 2.373$ is the matrix multiplication exponent [17, 10]). This matches the current best known bound for the simpler problem of RP for directed graph with weights $[-M,M]$, by Vassilevska Williams [16]. Quite interestingly, for integer edge weights in the range $[-M,M]$, the authors of [5] gave a higher upper bound of $\tilde{O}(M^{\frac{1}{4-\omega}}n^{2+\frac{1}{4-\omega}})$ time, which creates an interesting gap between the SSRP problem and the RP problem for negative

integer weights. Grandoni and Vassilevska Williams [5] conjectured that the gap between these two problems is essential and in fact they conjectured that the SSRP problem with negative weights is as hard as the directed APSP problem.

The algorithm described in [5] uses fast matrix multiplication tricks in order to break the trivial cubic upper bound, such algorithms are known as "algebraic algorithms". Algorithms that do not use any matrix multiplication tricks are known as "combinatorial algorithms". The interest in combinatorial algorithms mainly stems from the assumption that in practice combinatorial algorithms are much more efficient since the constants and sub-polynomial factors hidden in the matrix multiplication bounds are considered to be very high.

The SSRP problem was also recently considered in the combinatorial setting by Chechik and Cohen in [2] for undirected unweighted graphs. Specifically, Chechik and Cohen in [2] gave an $\tilde{O}(m\sqrt{n}+n^2)$ time randomized algorithm for SSRP in undirected unweighted graphs. Moreover, using conditional lower bounds Chechik and Cohen also showed that under some reasonable assumptions any combinatorial algorithm for the SSRP problem in unweighted undirected graphs requires $\tilde{\Omega}(m\sqrt{n})$ time.

Since there is little hope to solve the weighted case, the only missing piece in the picture of combinatorial SSRP is the case of directed unweighted graphs.

For the directed unweighted case it was shown earlier by Vassilevska Williams and Williams [18], using a conditional combinatorial lower bound that under some reasonable assumptions any combinatorial algorithm for the directed unweighted RP (and hence SSRP) problem requires $\tilde{\Omega}(m\sqrt{n})$ time. For the seemingly easier problem of replacement paths Roditty and Zwick [15] showed a near optimal solution of $\tilde{O}(m\sqrt{n})$ time for directed unweighted graphs.

Note that in the undirected unweighted case there is an essential gap between the RP and the SSRP problems. A natural question is whether such a gap also exists in the directed unweighted case. In this paper we show that this is not the case by providing a combinatorial near optimal $\tilde{O}(m\sqrt{n}+n^2)$ time algorithm for the case of directed unweighted graphs, which up to the n^2 factor (that is unavoidable as the output itself is of size $O(n^2)$) matches the running time of the algorithm in [15] (and also matches the running time of the undirected case in [2]). We therefore (up to poly-logarithmic factors) complete the picture of combinatorial SSRP.

Our main result is as follows.

▶ **Theorem 1.1.** *There exists an $\tilde{O}(m\sqrt{n}+n^2)$ time combinatorial algorithm for the SSRP problem on unweighted directed graphs. Our randomized algorithm is Monte Carlo with a one-sided error, as we always output distances which are at least the exact distances, and with high probability (of at least $1-n^{-C}$ for any constant $C>0$) we output the exact distance.*

Note that for unweighted directed graphs where $m=\tilde{O}(n^{1.5})$ our algorithm runs in $\tilde{O}(n^2)$ time, which is the time it takes just to output the result. Namely, in this range of density our algorithm surpasses the current best algebraic SSRP algorithm [5] (which has a running time complexity of $\tilde{O}(n^\omega)$) as long as $\omega>2$.

We will note that while we focus on the case of edge failures, in the directed case there is a well known reduction showing that edge failures can be used to simulate vertex failures. The reduction is as follows, replace every vertex v with two vertices v_{in} and v_{out}, and connect them by a direct edge $(v_{\text{in}},v_{\text{out}})$. Then, for every incoming edge (u,v) add the edge (u,v_{in}), and for every outgoing edge (v,u) add the edge (v_{out},u). The failure of the vertex v is now simulated by the failure of the edge $(v_{\text{in}},v_{\text{out}})$.

Our main novelty is in the introduction of a tool which we refer to as *weight functions*. This tool proved to be very useful in order to apply a divide and conquer approach and could perhaps be utilized in other related problems.

1.1 Rational Weights

While we describe an algorithm for the problem of SSRP in unweighted graphs, our algorithm (much like the directed RP algorithm [15]) can be easily generalized to solve the case of weighted graphs for **rational** edge weights in the range $[1, C]$, for every constant $C \geq 1$, in the same time complexity. This is because the only place our algorithm (and the algorithm from [15]) uses the fact that the graph is unweighted is in the claim that a path of length l contains $\Theta(l)$ vertices, which is used in order to utilize sampling techniques. As this is also true for rational weights in the range $[1, C]$, our algorithm generalizes for this case trivially.

Algebraic algorithms inherently can not perform on graphs with rational weights. This is since algebraic algorithms use a reduction from a problem known as min-plus product [2] to the problem of matrix product, and this reduction works only for **integer** weights. Since in some use-cases (like the k-simple paths problem) it is very useful to have rational weights, this shows another potential interest in combinatorial algorithms.

We note that in order to store **rational** numbers, we must make some common assumptions regarding the model of computation. More specifically, we assume that computing the summation of n edge weights can be performed in $\tilde{O}(n)$ time and that all numbers we are dealing with can be stored in one (or $O(1)$) space unit. A realistic option is working in a word-RAM model, and considering only rational edge weights which are of the form $\frac{m}{2^k}$, where the two integers m and 2^k fit in the size of $O(1)$ computer words. This way, the summation of $O(n)$ numbers also fits in $O(1)$ computer words. This way of representing rational numbers is reminiscent of the floating-point representation, that is commonly used in practical applications.

In Section 7 in the full version of this paper, we show that any algorithm (combinatorial or not) for the SSRP problem for graphs with rational edge weights from the range $[1, 2)$, that runs in $O(mn^{1/2-\epsilon})$ time for any fixed $\epsilon > 0$ implies a truly sub-cubic algorithm for APSP over graphs with arbitrary integer weights. The claim is formally stated in Theorem 7.1. Previously such a conditional lower bound was only known for combinatorial algorithms using a reduction from Boolean Matrix Multiplication (see [2]).

2 Preliminaries

We will use the following notation: $\mathbb{N} = \{0, 1, 2, ...\}, \mathbb{N}^+ = \{1, 2, 3, ...\}, \forall a \in \mathbb{N}^+ : [a] = \{1, 2, ..., a\}$. Let G be a weighted directed graph then $E(G)$ denotes the set of edges in G and $V(G)$ the set of nodes. For a vertex v we say that $v \in G$ if $v \in V(G)$ and for an edge e we say that $e \in G$ if $e \in E(G)$. Let $u, v \in V(G)$ be two vertices, we denote by $d(u, v, G)$ the distance from u to v in the graph G, and denote by $R(u, v, G)$ **some** shortest path from u to v in G. Let $P \subseteq G$ be a path from u to v, we define $|P| = |E(P)| = |V(P)| - 1$. We also denote the length of P by $d(P)$. Note that $d(R(u, v, G)) = d(u, v, G)$. For a set of edges $A \subseteq E(G)$ we denote the graph $(V(G), E(G) \setminus A)$ by $G - A$. For an edge e we shortly denote $G - \{e\}$ by $G - e$, and for a path P we shortly denote $G - E(P)$ by $G - P$.

We denote by G^R the graph obtained by reversing the directions of all edges - that is the graph obtained by replacing each edge $(v, u) \in E(G)$ with the edge (u, v) with the same weight. Given a sub-graph $H \subseteq G$ we denote by $G[H]$ the sub-graph of G induced by the nodes in $V(H)$.

[2] Also known as funny matrix multiplication or distance product, see [1, 20]

Let $P \subseteq G$ be a shortest path from a node s to a node t. Let $u, v \in V(P)$ be two nodes in P, we say that u is **before** v in P if $d(u,t,G) \geq d(v,t,G)$ and that u is **after** v in P if $d(u,t,G) \leq d(v,t,G)$, For an edge $e = (u,v) \in E(P)$ and a node $a \in V(P)$ we say that a is **before** e in the path P if a is **before** u in P and say that a is **after** e in P if a is **after** v in the path P.

The following sampling Lemma is a folklore.

▶ **Lemma 2.1** (Sampling Lemma). *Consider n balls of which R are red and $n - R$ are blue. Let $C > 0, N > 0$ be two numbers such that $R > C \cdot \ln(N)$. Let B be a random set of balls such that each ball is chosen to be in B independently at random with probability $\frac{C \cdot \ln(N)}{R}$. Then w.h.p (with probability at least $1 - \frac{2}{N^c}$) there is a red ball in B and the size of B is $\tilde{O}(n/R)$.*

The following separation Lemma was used extensively in many divide and conquer algorithms on graphs including the algebraic SSRP algorithm from [5].

▶ **Lemma 2.2** (Separator Lemma). *Given a tree K with n nodes rooted at a node s, one can find in $O(n)$ time a node t that separates the tree K into 2 edge disjoint sub-trees S, T such that $E(S) \cup E(T) = E(K)$, $V(T) \cap V(S) = \{t\}$ and $\frac{n}{3} \leq |V(T)|, |V(S)| \leq \frac{2n}{3}$*

WLOG we always assume that $s \in V(S)$, which implies that t must be the root of T. Note that it might be the case that $t = s$.

2.1 The Generalized SSRP Problem

We next describe a generalization of the directed-SSRP problem that our algorithm works with. We start by describing the notation of weight functions, a new concept we developed that allows us to compress a lot of information into one recursive call of the algorithm. In the next section we will give more intuition about the weight functions and this specific generalization.

▶ **Definition 2.3** (Weight Function). *Let G be an **unweighted** directed graph. Let $s \in V(G)$ be some special source node. A function $w : V(G) \to \mathbb{N} \cup \{\infty\}$ is a weight function (with respect to the source node s) if $w(v) \geq d(s,v,G)$ for every vertex $v \in V(G)$. We refer to this requirement as the weight requirement.*

For a source node $s \in V(G)$ and a weight function w (with respect to the source node s) we define the **weighted** directed graph G_w by taking the unweighted graph G, and assigning each edge the weight 1. We then add for every node $v \in V(G)$ the edge (s,v) and assign to it the weight $w(v)$. Note that G is a sub-graph of G_w. Also, note that by the weight requirement, for every two nodes $u, v \in V(G) : d(u,v,G) = d(u,v,G_w)$.

The generalized SSRP problem is now defined as follows. The input consists of the following:

- An unweighted directed graph H and a source vertex $s \in V(H)$
- A BFS tree K in H rooted at the source s ($E(K) \subseteq E(H)$)
- A set of weight functions W (with respect to source node s)
- A set of queries $Q \subseteq E(K) \times V(H) \times W$

The goal is to output for every $(e,x,w) \in Q$ the distance $d(s,x,H_w - e)$. Note that this problem is indeed a generalization of the classic SSRP problem. In order to solve the SSRP problem on the initial graph G and the BFS tree $\widehat{K}$, we simply define a single weight function $w : V(G) \to \mathbb{N} \cup \{\infty\}$ that is defined to be $w \equiv \infty$. We then invoke our algorithm

with the graph G, the BFS tree $\widehat{K}$, the set of weight functions $W = \{w\}$, and the query set $Q = E(\widehat{K}) \times V(G) \times W$. Note that G and G_w are the same graph in the sense that for every edge $e \in E(G)$ and destination $x \in V(G)$ we have that $d(s, x, G - e) = d(s, x, G_w - e)$. Hence, invoking our algorithm for the generalized SSRP would suffice. As we will only work with the generalized SSRP problem, we here after refer to it as the SSRP problem for simplicity.

3 Overview

Our algorithm uses a divide and conquer approach. Each recursive call works on a different sub-tree (K) of the original BFS tree ($\widehat{K}$), where both the destination and the edge failure are within this sub-tree (for the case when the edge failure and the destination are not in the same sub-tree our algorithm solves this in a non recursive manner to be described later in case 1 of the algorithm overview). The vertices of the sub-tree K induce a sub-graph H of the original graph G. Denote by $n = |V(G)|, m = |E(G)|, n_H = |V(H)|, m_H = |E(H)|$.

The first step of our algorithm is to separate the input BFS tree K into two edge disjoint sub-trees S and T using a balanced tree separator (see Lemma 2.2). We denote the root of the BFS tree K by s. We assume WLOG that the root of S is s and the root of T is some node t. It might be the case that $s = t$. We define P as the path from s to t in the BFS tree K. Note that $P \subseteq S$. An illustration of this separation can be found in Figure 1.

Let K' be one of the two sub-trees of K (that is S or T). If a replacement path is fully contained in the graph induced by K' then simply using the recursive call is enough in order to compute its length. The more challenging case is when the replacement path contains vertices that are not in K'.

In [5] the authors used a somewhat similar divide and conquer approach. Consider a recursive call on a sub-tree K' and consider the case when the edge failure and destination node are both in K'. In their algorithm, the authors of [5] used sampling techniques and a truncated version of the algebraic APSP algorithm (as presented in [21]) in order to create a compressed version of the subgraph induced over K' (by adding shortcuts between vertices in K'), which (w.h.p) preserves all information needed in order to compute the true distance.

However, in the combinatorial setting, one cannot use this sort of compression process for several reasons. Firstly, after the first call to the compression step (as described in the algorithm in [5]), the resulted graph could be very dense, maybe even complete. Since the conditional lower bound of $\tilde{\Omega}(m\sqrt{n} + n^2)$ for a combinatorial SSRP ([2, 18]) depends on the number of edges, we do not want to receive such dense graphs. Secondly, as we are in the combinatorial setting, we cannot use fast matrix multiplication in the compression step, which is a critical part of the algorithm described in [5]. Lastly, after the compression step the resulted graph is weighted which leaves us with a substantially more difficult problem. In fact in the combinatorial setting there is no sub-cubic time algorithm that solves the even seemingly easier problem of weighted RP (see [18] for conditional lower bounds).

So in the combinatorial setting we must devise a new, more restricted, compression technique. We will essentially show that if we add weighted edges only from the source s to all other vertices, and restrict the weights to be such that the weight of the edge (s, v) is at least $d(s, v, H)$, then solving replacement path on such a graph still requires only $\tilde{O}(m_H\sqrt{n_H} + n_H^2)$ time. We therefore would like to add only edges between s and all other nodes. However, this quickly proves to be difficult, and it seems that if we add only weighted edges from s to the compressed graph we either "under-shoot" and do not represent all replacement paths, or we "over-shoot" and represent replacement paths that does not really exist in the graph $H - e$ (for some edge failure e) - such paths will be called untruthful paths.

We have devised a technique to fix the over-shooting. That is, we give the recursive call weights that may represent untruthful replacement paths in $H - e$, but we force the recursive call to restrict the replacement paths it searches for, so we will be able to fix them before the algorithm outputs them, while maintaining optimality. The way we do so is by a novel concept we call weight functions. The idea is that the unweighted graph H will come equipped with a set of functions W, such that every $w \in W$ is a function from $V(H)$ to $\mathbb{N} \cup \{\infty\}$ and for every vertex v it holds that $w(v) \geq d(s, v, H)$. For every weight function $w \in W$ the weighted graph H_w is defined by adding for every node $v \in V$ the edge (s, v) with weight $w(v)$. The goal of the algorithm is then outputting $d(s, v, H_w - e)$ for every triplet $x \in V, w \in W, e \in E(K)$. By restricting the algorithm to only use a single, specific weight function we achieve enough "control" to fix the untruthful paths. In order to maintain the desired running time it will be critical to keep the number of weight functions ($|W|$) at most $\tilde{O}(\sqrt{n})$ (where n is the number of nodes of the original graph G).

3.1 Algorithm Overview

In the remainder of this section we sketch the ideas of our algorithm in high level. For the sake of simplicity, the algorithm in this section runs in $\tilde{O}(n^{2.5})$ time rather than $\tilde{O}(m\sqrt{n} + n^2)$ time. At the end of this section we will briefly describe how one can use some simple techniques to reduce the running time to the near optimal of $\tilde{O}(m\sqrt{n} + n^2)$. While sketching the algorithm, we also ignore the query set Q, as it is only necessary when reducing the running time of the algorithm to $\tilde{O}(m\sqrt{n} + n^2)$. So the goal of the algorithm in this section is to estimate $d(s, x, H_w - e)$ **for every** $e \in E(K), x \in V(H)$ and $w \in W$. The complete algorithm and proof of correctness can be found in Sections 4 and 5 in the full version of this paper.

In our algorithm we distinguish between a few cases according to where the edge failure and the destination are with respect to S and T. Note that for each edge failure e and destination node x we clearly know in which case we are. In each such case we distinguish between different sub-cases according to different properties of the replacement path. Clearly we do not know the replacement path a-priori, meaning that we do not know in which sub-case we are. So when proving the correctness of our algorithm in Section 5 we show that the estimation created for every sub-case is always at least the real value of $d(s, x, H_w - e)$, that is, we do not underestimate. Then we show that for the true sub-case (the sub-case describing the true replacement path) our estimation matches the true value of $d(s, x, H_w - e)$ w.h.p. By returning the minimum estimation from all of the sub-cases we are guaranteed to return the true distance w.h.p. To distinguish between the different sub-cases we first define two useful characterization of replacement paths in H_w.

▶ **Definition 3.1** (Weighted paths). *Let $e \in E(K), x \in V(H), w \in W$, and let R be a path from s to x in the graph $H_w - e$. The path R will be called weighted if it uses some edge from $E(H_w) - E(H)$. R will be called unweighted if it is fully contained in $H - e$.*

The following crucial observation allows us to handle many cases involving weighted replacement paths

▶ **Observation 3.2.** Let $e \in P$ be an edge failure, $x \in H$ be a destination node and $w \in W$ be a weight function. If the replacement path $R(s, x, H_w - e)$ is weighted then it leaves P at s and does not intersect with P until **after** the edge failure.

To see why this observation is true, first note that all the edges in $E(H_w) - E(H)$ begin at s by definition, so $R(s, x, H_w - e)$ indeed leaves P at s. Also, $R(s, x, H_w - e)$ does not intersect with P until after the edge failure as otherwise $R(s, x, H_w - e)$ could have used

the path P to get from s to the intersection point, which is a shortest path by the weight requirements. In other words, the use of the weighted edge is unnecessary. An illustration of such path can be seen in Figure 6.

For edge failures from P we also define the following useful characterization

▶ **Definition 3.3** (Jumping and Departing Paths). *Let $e \in E(P), x \in V(H), w \in W$, and let R be a path from s to x in the graph $H_w - e$. The path R will be called jumping if it uses some node u such that $u \in P$ and u is **after** the edge failure e in the path P. A path that is not jumping will be called departing.*

First case – the failure is in P and the destination is in T

This case can be solved in a non-recursive manner, using observation 3.2 and somewhat similar observations to those that were used in [5]. We distinguish between 3 different forms the path $R(s, x, H_w - e)$ can take:

Case 1.1: $R(s, x, H_w - e)$ is departing and weighted.

Using observation 3.2, we can conclude that $R(s, x, H_w - e)$ is edge-disjoint from P as it does not intersect with P before the edge failure nor after (since it is departing). This implies that the length of $R(s, x, H_w - e)$ is $d(s, x, H_w - P)$. This value can easily be computed by running Dijkstra's algorithm from s in the graph $H_w - P$ for every weight function w.

Case 1.2: $R(s, x, H_w - e)$ is departing and unweighted.

An illustration of this case can be seen in Figure 3. In this case one can use a technique similar to the one used in [5] in order to compute length of the replacement path w.h.p. That is, if e is among the last $\sqrt{n_H}$ edges of P, then we can use a brute force solution to compute $d(s, x, H - e)$. If e is of distance at least $\sqrt{n_H}$ from t, then the length of the detour of $R(s, x, H_w - e)$ is at least $\sqrt{n_H}$ as this path departs before e and gets to T. So by sampling a set of nodes B of size $\tilde{O}(\sqrt{n_H})$, we hit every such detour w.h.p. Assuming we hit the detour using the pivot node $b \in B$, we can compute $d(s, b, H - e)$ rather easily, and have that $d(s, x, H - e) = d(s, b, H - e) + d(b, x, H - P)$.

In the full algorithm we denote the estimation obtained by the pivots sampling by $\text{Depart}(s, x, e)$. We show how to compute this estimation in step 5 of the full algorithm and prove its correctness in Claims 5.1 and 5.2.

Case 1.3: $R(s, x, H_w - e)$ is jumping.

As observed by the authors of [5], taking care of jumping replacement paths in the case when $x \in T$ essentially reduces to solving the RP problem, where the source node is s and the destination node is t. This is since a jumping replacement path passes WLOG through the separator node t.

So we focus on computing the length of $R(s, t, H_w - e)$ for every $e \in P$ and $w \in W$. Using observation 3.2, if $R(s, t, H_w - e)$ is weighted then its length is $\min_{u \text{ after } e \text{ in } P}\{d(s, u, H_w - P) + d(u, t, H)\}$ as it does not intersect with P until after the edge failure and from the intersection node the replacement path can go to t using the shortest path P (as this subpath does not contain the edge failure e). Computing this value naively for every $w \in W$ and $e \in P$ takes $\tilde{O}(|W|{n_H}^2)$ time.

If $R(s, t, H_w - e)$ is unweighted, the algorithm of [15] can be used to compute its length.

Second case – the failure is in T and the destination is in T

We solve this case recursively. The recursive call will be invoked over the subgraph $H[T]$. Because the root of the tree T is t and not s, we must change the source of our SSRP. This implies that replacement paths that use the path P to get from s to t will be $d(s,t,H)$ units shorter in the recursive call than they truly are. So when we compress different forms of replacement paths using weight functions, for normalization reasons we must also subtract $d(s,t,H)$ from the weight function. For simplicity, we ignore this issue in the overview, but keep in mind that we always need to subtract $d(s,t,H)$ from every weight function before the algorithm passes them to the recursion call, and add this value back when it receives the recursion's estimation.

We distinguish between two possible forms of the replacement path: weighted and unweighted. Rather interestingly we will see that this separation provides enough information about the structure of the replacement path in order to compress it, and find its length recursively.

Case 2.1: The path $R(s,x,H_w-e)$ is weighted.

We claim that in this case, the only node from S that $R(s,x,H_w-e)$ uses is s. To see this note that for every node u from S that is not s, the path from s to u in the BFS tree K does not contain the edge failure e, since $e \in T$. By the weight requirement this path is a shortest path in H_w. Hence, if a weighted replacement path uses a node u from S, it can use the path from s to u in K. In other words, the use of a weighted edge was unnecessary. So in this case the replacement path is almost completely contained within $H[T]$. Therefore, in order to take care of this case, we simply need to pass the weight function w to the recursive call. We formally prove the correctness of this case in Claim 5.20.

Case 2.2: The path $R(s,x,H_w-e)$ is unweighted.

Let u be the last node of $R(s,x,H_w-e)$ that is from S. If u is t, then we can separate $R(s,x,H_w-e)$ into two subpaths: a path from s to t - that is the shortest path P WLOG, and the shortest path from t to x in $H[T]-e$. We can use the recursive call over $H[T]$ to compute the length of the second sub-path, and when we add $d(s,t,H)$ we will get the length of $R(s,x,H_w-e)$.

The more interesting case is when $u \neq t$. Let v denote the node right after u on $R(s,x,H_w-e)$. In this case we say that v gets "helped from above" by u, as illustrated in Figure 8. Since u is the last node in $R(s,x,H_w-e)$ that belongs to S the sub-path of $R(s,x,H_w-e)$ from v to x is fully contained in $H[T]-e$. So we only need to compress the sub-path of $R(s,x,H_w-e)$ from s to v. In order to do so we define a new weight function $c_T : V(T) \to \mathbb{N} \cup \{\infty\}$ where for every vertex v , $c_T(v)$ is defined to be $\min_{u \in V(S)-\{t\}:(u,v)\in E(H)} \{d(s,u,H)+1\}$. The sub-path of $R(s,x,H_w-e)$ from s to v is represented in the graph $H[T]_{c_T}-e$ as the weighted edge (t,v). So by passing c_T to the recursion and computing $d(t,x,H[T]_{c_T}-e)$ we will be able to obtain the length of the replacement path. We formally prove the correctness of this case in Claim 5.19.

We note that c_T is truthful, in the sense that for every edge failure $e \in T$, $c_T(v)$ is the length of some path from s to v in $H-e$. This is since the path from s to u in K is of length $d(s,u,H)$ and does not contain e (as previously claimed), and the edge (u,v) is of length 1 and is not in T because u is not in T. We formally prove that c_T is truthful as part of Claim 5.18.

Third case – the failure is in $E(S) - E(P)$ and the destination is in S

We handle this case similarly to the way we handled the second case. However we still sketch the algorithm for this case as it will introduce the notation of a "help from bellow" replacement path, which will be useful in the fourth case. We solve this case recursively. The recursive call will be invoked over the subgraph $H[S]$. In order to take care of this case we distinguish between two forms of the replacement path $R(s, x, H_w - e)$.

Case 3.1: The path $R(s, x, H_w - e)$ uses only nodes from S.
In this case simply passing the weight function w to the recursive call would suffice in order to compute the length of $R(s, x, H_w - e)$.

Case 3.2: $R(s, x, H_w - e)$ uses a node from $V(T) - \{t\}$.
Let u be the last node in $R(s, x, H_w - e)$ that belongs to $V(T) - \{t\}$. Note that the path from s to u in the BFS tree K uses only edges from P and T, meaning that it does not use the edge failure $e \in E(S) - E(P)$. Hence, WLOG we may assume that the sub-path from s to u in $R(s, x, H_w - e)$ is the path from s to u in the BFS tree K as this is a shortest path by the weight requirements. Note that this implies in particular that $R(s, x, H_w - e)$ is unweighted. An illustration of this case can be found in Figure 4. We name this kind of paths "help from bellow" replacement paths. Let v be the node right after u in $R(s, x, H_w - e)$. Since u was the last node in $R(s, x, H_w - e)$ that belongs to $V(T) - \{t\}$ the sub-path of $R(s, x, H_w - e)$ from v to x is fully contained in $H[S] - e$.

So we only need to compress the sub-path of $R(s, x, H_w - e)$ from s to v. In order to do so we define a new weight function $c_S : V(S) \to \mathbb{N} \cup \{\infty\}$ where for every vertex v , $c_S(v)$ is defined to be $\min_{u \in V(T)-\{t\}:(u,v)\in E(H)} \{d(s, u, H) + 1\}$. The sub-path of $R(s, x, H_w - e)$ from s to v is represented in the graph $H[S]_{c_S}$ by the weighted edge (s, v). So by passing the weight function c_S to the recursive call over $H[S]$, and computing $d(s, x, H[S]_{c_S} - e)$, we will be able to compute the length of $R(s, x, H_w - e)$. We formally prove the correctness of this case in Claim 5.23.

We also claim that this function is truthful for edge failures from $E(S) - E(P)$ in the sense that for every $e \in E(S) - E(P)$, $c_S(v)$ is the weight of some path from s to v in $H - e$. This is since the path from s to u in K is of length $d(s, u, H)$ and does not contain e (as previously claimed), and the edge (u, v) is of length 1 and is not in S because u is not in S. We formally prove this fact as part of Claim 5.22.

Note that the weight function c_S is untruthful for edge failures from P, as the path from s to u in K contains the entire path P. But if we consider the recursion's estimation for $d(s, x, H[S]_{c_S} - e)$ **only** for an edge failure $e \in E(S) - E(P)$, we are promised that this estimation represents the length of a true path in $H - e$. If we were to add weighted edges instead of weight functions, we would lose the ability to consider $d(s, x, H[S]_{c_S} - e)$ as an estimation for $d(s, x, H_w - e)$ only for specific edge failures.

Fourth case – the failure is in P and the destination is in S

This case is the most complicated case in our algorithm. Since we cannot allow three recursive calls (in order to obtain the desired running time) and because we see no efficient way to solve this case in a non-recursive manner, we will need to use the same recursive call over $H[S]$ as in the previous case (the third case). We will do so by adding more weight functions.

We begin by making two simple observations that take care of some easy cases, so we could focus on the more involved ones.

- If the replacement path $R(s, x, H_w - e)$ uses no nodes from T then one can simply use a recursive call over the graph $H[S]$ to compute its length.
- If $R(s, x, H_w - e)$ is departing, then since we may assume it contains nodes from T, we can use observations similar to those made in cases (1.1) and (1.2) in order to compute its length.

So we now focus on the more interesting case when $R(s, x, H_w - e)$ uses nodes from T and is jumping. Note that since $R(s, x, H_w - e)$ is jumping it must leave the path P at some node v_i before the edge failure and return to P at some node v_j after the edge failure. We will in fact still need to separate this case into 3 further sub-cases, depending on the order $R(s, x, H_w - e)$ uses nodes from T. These 3 cases present the true power of weight functions, and their ability to compress graphs in a way that is sometimes untruthful but fixable.

Case 4.1: $R(s, x, H_w - e)$ uses a node from T after it uses v_j. An illustration for this case can be found in Figure 5.

We claim that in this case the length of $R(s, x, H_w - e)$ is $d(s, x, H[S]_{c_S} - e) - d(s, t, H) + d(s, t, H_w - e)$. While formally proving the correctness of this claim is rather technical we attempt to give some intuition for this claim. Note that since $R(s, x, H_w - e)$ uses a node from T after it uses v_j, it passes WLOG through t (as v_j is after the edge failure). So we can split $R(s, x, H_w - e)$ into two sub-paths: the replacement path from s to t - which is of length $d(s, t, H_w - e)$, and the path from t to x - which we denote by $R[t, x]$.

Lets us consider the path $P \circ R[t, x]$. We claim that even though the path $R[t, x]$ contains nodes from T, the recursive call over $H[S]$ can evaluate the length of the path $P \circ R[t, x]$. This is because, roughly speaking, the path $P \circ R[t, x]$ is a sort of "help from bellow" replacement path - as described in the the third case in which $e \in E(S) - E(P)$. So like in the "help from bellow" case, the path $P \circ R[t, x]$ would be represented in $H[S]_{c_S} - e$ as a weighted replacement path. When we receive the length of this weighted replacement path we remove P and replace it with the replacement path from s to t. That is, we subtract $d(s, t, H)$ and add $d(s, t, H_w - e)$. We formally prove the correctness of this estimation in Claim 5.12.As stated in the beginning of the overview, we do not know a-priori if the replacement path indeed falls in this sub-case, so we have to make sure that we never underestimate $d(s, x, H_w - e)$. We formally prove this in Claim 5.6.In this case we see that weight functions allow us to assign weights that are untruthful for some edge failures, but give us enough control in order to fix the untruthful replacement paths.

Note that $d(s, x, H[S]_{c_S} - e)$ is used regardless of which weight function w the true replacement path uses. The fact that we use one recursive call over all weight functions, allows us to compute this term only once, which we could not do if the algorithm would have used a different recursive call for each weight function.

Case 4.2: $R(s, x, H_w - e)$ is weighted and it uses no nodes from T after v_j. An illustration for this case can be found in Figure 6.

Let (s, v) be the weighted edge in the replacement path $R(s, x, H_w - e)$. Note that (s, v) is not in P (as P contains only unweighted edges from H). Hence, by definition the replacement path leaves the path P at s, that is, $v_i = s$.

This implies that the sub-path from s to v_j is edge disjoint to P and so its length is $d(s, v_j, H_w - P)$. So for every weight function $w \in W$, we would have wished to define a new weight function $w|_P$ such that $w|_P(v_j) = d(s, v_j, H_w - P)$ for every $v_j \in P$. We will then

ask the recursive call to estimate $d(s, x, H[S]_{w|_P} - e)$. This will indeed suffice in order to compute the length of the replacement path recursively, as $R(s, x, H_w - e)$ uses no nodes from T after v_j.

However, by doing so we increase the number of weight function passed to the recursive call by a factor of 2. This sort of exponential growth will prevent us from achieving the desired running time. So instead we define a new weight function $w|_S$ such that $w|_S(x) = d(s, x, H_w - P)$ if $x \in P$ and $w|_S(x) = w(x)$ if $x \notin P$. Note that for every x it holds that $w|_S(x) \leq w(x)$, since the distance $d(s, x, H_w - P)$ is at most the weight of the edge $(s, x) \in H_w$ which is $w(x)$. This implies that the $w|_S$ function preserves information from both w and $w|_P$. So instead of passing w to the recursive call, we pass $w|_S$. Later in Claim 5.5 we prove that the new $w|_S$ function is truthful in the sense that for every $e \in S, x \in S$ it holds that $d(s, x, H[S]_{w|_S} - e)$ is at least $d(s, x, H_w - e)$, meaning we do not create underestimations by using $w|_S$ instead of w. In the full version of the algorithm, we prove the correctness of this case in Claim 5.13.

So as one can see, weight functions allow us to specifically choose special nodes and decrease their weights in order to compress more information, without sacrificing the truthfulness of the weight function.

Case 4.3: $R(s, x, H_w - e)$ is unweighted, it uses no nodes from T after v_j.

This is the most involved and interesting case our algorithm handles. Note that since we assume $R(s, x, H_w - e)$ uses a node from T, and since $R(s, x, H_w - e)$ uses no nodes from T after v_j, then the sub-path of $R(s, x, H_w - e)$ from v_i to v_j must contain a node from T. An illustration for this case can be found in Figure 7. Similarly to Case 1.2, we may assume that the edge failure is not among the last $\sqrt{n_H}$ edges of P as otherwise we can use a brute force solution to compute the length of the replacement path. Since the sub-path from v_i to v_j uses a node from T, its length is at least $d(v_i, t, H)$ that is at least $\sqrt{n_H}$. So w.h.p we have sampled some pivot node $b \in B$ on this sub-path. Note that the sub-path from s to b is departing as the replacement path returns to P only at v_j. So we can easily compute $d(s, b, H - e)$ as stated before in Case 1.2.

To compress the sub-path from b to v_j we define a weight function w_b for every pivot node. We would have wished to define $w_b(v_j) = d(b, v_j, H - P)$, recursively compute $d(s, x, H[S]_{w_b} - e)$ and add $d(s, b, H - e)$ when receiving the answer from the recursion. This will indeed suffice in order to compress the length of the sub-path from v_i to v_j as it is edge disjoint to P. However this is not a valid weight function as it does not necessarily fulfill the weight requirements. So instead we define $w_b(v_j) = d(s, b, H) + d(b, v_j, H - P)$ which is a valid weight function, and we fix the output of the recursion by replacing $d(s, b, H)$ with $d(s, b, H - e)$, i.e. subtracting the former and adding the latter. As in case 4.1, we need to prove that we never underestimate $d(s, x, H - e)$. This is formally done in Claim 5.7.

As we can see, while the weight function w_b is untruthful, in the sense that $w_b(v_j)$ is not necessarily the distance of a path from s to v_j in $H - e$, we are able to fix this untruthfulness as we know what pivot $b \in B$ is used in each weight function w_b. In a sense if we could have used a different recursive call for every $b \in B$ we could have used edges from s rather than weight functions but this would be very inefficient. The weight functions allow us to compress all these recursive calls into one.

In the full version of the algorithm we denote the estimation made using the w_b functions by $\text{Pivot}(s, x, e)$. We show how to compute this estimation in step 9 of the algorithm and prove the correctness of this case in Claim 5.14.

3.2 Running Time Analysis

First we note that the number of weight functions in each recursive call increases by $\tilde{O}(\sqrt{n_H})$ in each level of the recursion, as we add the c_S, c_T and $\{w_b\}_{b\in B}$ weight functions, and $|B| = \tilde{O}(\sqrt{n_H})$. Since the number of the weight functions in the first call to the algorithm is 1 (the function $w \equiv \infty$), and since the depth of the recursion is logarithmic we have that $|W| = \tilde{O}(\sqrt{n})$ at all times. So if we simply analyze the non-recursive parts of the algorithm, we can conclude the algorithm spends $\tilde{O}(n_H^2\sqrt{n})$ times on the recursive call over the sub-graph H. One can rather easily see that since the BFS trees in each level of the recursion are edge disjoint sub-trees of the original BFS tree $\widehat{K}$, the total number of vertices in each level of the recursion is at most $2n$. We prove this formally in Section 6. So the total time the algorithm spends on each level of the recursion is $\tilde{O}(n^{2.5})$. Since the depth of the recursion is logarithmic the running time of the algorithm is $\tilde{O}(n^{2.5})$.

3.3 Going From $\tilde{O}(n^{2.5})$ to $\tilde{O}(m\sqrt{n}+n^2)$

In this section we sketch the ideas of improving the running time from $\tilde{O}(n^{2.5})$ to $\tilde{O}(m\sqrt{n}+n^2)$.

We first note that as the number of weight functions in our algorithm is $\tilde{O}(\sqrt{n})$ in each recursive call then even outputting $d(s,x,H_w-e)$ for every triplet $w \in W, x \in V(H), e \in E(K)$ is impossible (in the desired running time) as there are $\tilde{O}(n_H^{2.5})$ such triplets. In order to overcome this issue, the algorithm does not output distances to all such triplets but rather each recursive call is given as input a set of queries Q that is a small subset of all possible triplets (that is $Q \subseteq E(K) \times V(H) \times W$) and the goal is to output the distances only for the given set of queries. Initially, Q is set to be $E(\widehat{K}) \times V(G) \times \{w\}$, where $w \equiv \infty$, and so its size is $|Q| = \tilde{O}(n^2)$. Each recursive call over a graph H will make sure to ask only $\tilde{O}({n_H}^2)$ new queries (queries which it didn't received). Since the total number of vertices in each level is at most $2n$, the number of new queries added at each level of the recursion is $\tilde{O}(n^2)$. Since the depth of the recursion is logarithmic, the total number of queries asked is $\tilde{O}(n^2)$.

Secondly, recall that in Case 1.3, where the edge failure is in $E(P)$ and the destination is t, the algorithm computes the value of $\min_{u \text{ is after } e \text{ in } P}\{d(s,u,H_w-P)+d(u,t,H)\}$ naïvely for every $e \in E(P), w \in W$. This computation costs $\tilde{O}(|W|n_H^2)$ time. However for a specific function $w \in W$, this value can be computed for all $e \in E(P)$ in $\tilde{O}(n_H)$ time using a simple dynamic programming argument which will be shown in Section 5.2 in the complete algorithm. Hence, we can reduce the running time of this part to $\tilde{O}(|W|n_H)$.

Finally, and most importantly, when handling departing unweighted paths (Case 1.2) and when using the w_b weight function (Case 4.3) the algorithm samples a set B of pivots of size $\tilde{O}(\sqrt{n_H})$. Then for every edge failure $e \in P$ and destination node $x \in V(H)$ we iterate over B and find the pivot that provides the smallest distance estimation. This implies that the algorithm spends $\tilde{O}(|B||P||V(H)|)$ time to find these pivots, which is again $\tilde{O}(n_H^{2.5})$ time. The problem is that our estimation for the distance between an edge failure and the separator node t is too loose. On the one hand when sampling B we say that this distance is at least $\sqrt{n_H}$, but on the other hand when bounding $|P|$ we say that it is at most $O(n_H)$.

In order to solve this problem we use a standard scaling trick. More specifically, we consider a logarithmic number of sub-paths $\{P_k\}$, where P_k is the sub-path of P induced by the vertices $\{v \in V(P) : 2^{k+1}\sqrt{n_H} \geq d(v,t,H) \geq 2^k\sqrt{n_H}\}$. P_0 is defined to be the sub-path of P induced by the last $2\sqrt{n_H}$ vertices of P. Note that the set of paths $\{P_k\}$ is an edge disjoint partition of P, and that $|P_k| = O(2^k\sqrt{n_H})$. An illustration for this partition can be seen in Figure 2. For every index $k \neq 0$ we then sample a random set B_k of size $\tilde{O}(\frac{\sqrt{n_H}}{2^k})$

using the sampling lemma 2.1. Now, if we consider an edge failure $e \in P_k$ for $k \neq 0$, we know that the distance from e to t is at least $2^k\sqrt{n_H}$. So when we wish to estimate the length of the departing replacement path in Case 1.2 or send the query (e, x, w_b) to the recursive call over $H[S]$ in Case 4.3, we only need consider pivot nodes b that are from B_k.

Figures

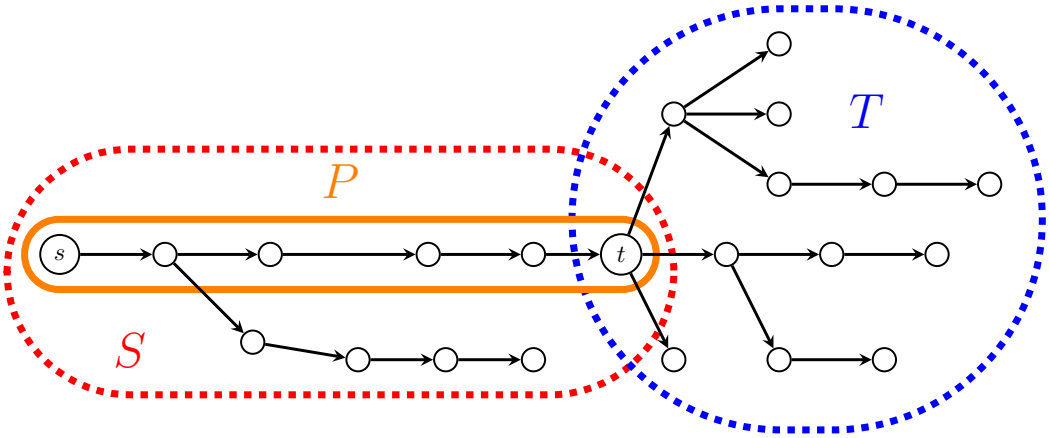

Figure 1 Tree separation.

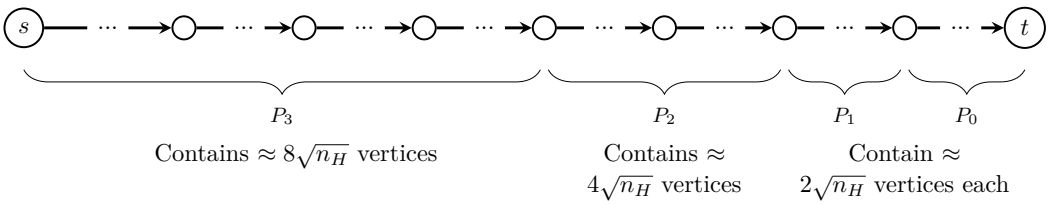

Figure 2 The P_k partition.

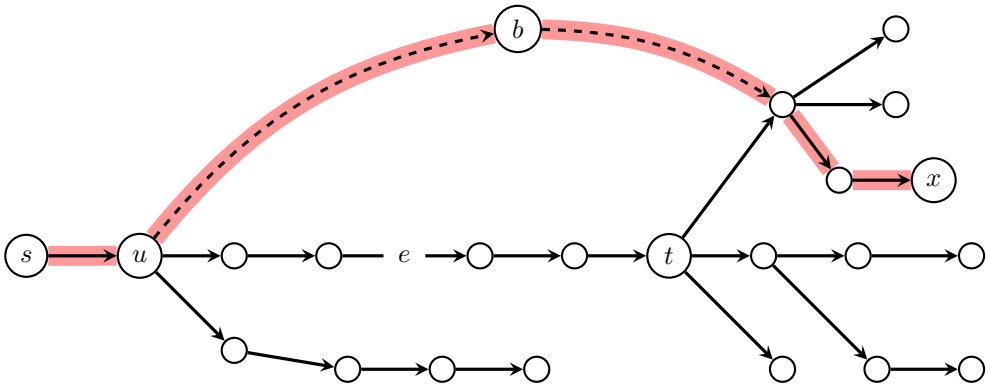

Figure 3 Departing replacement path in H.

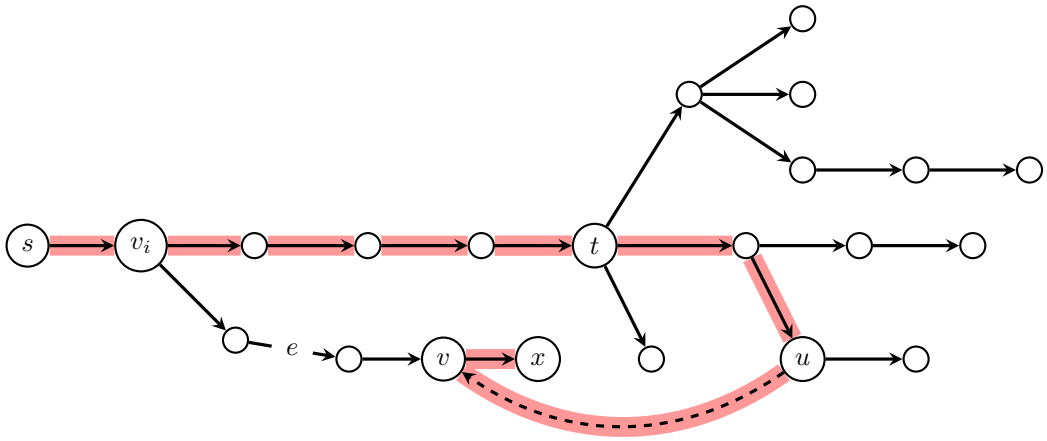

Figure 4 $R(s, x, H_w - e)$ is a "help from below" path.

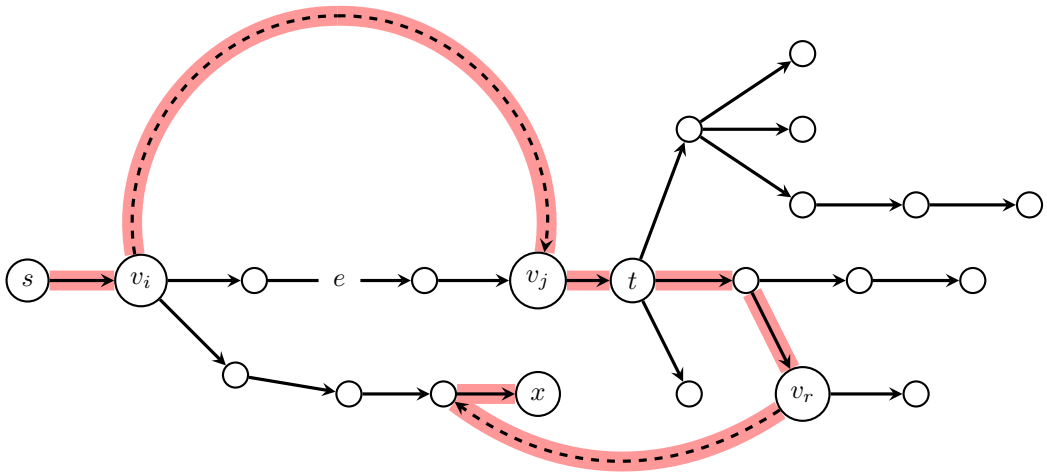

Figure 5 Case 4.1: $R(s, x, H_w - e)$ uses a node from T after it uses v_j.

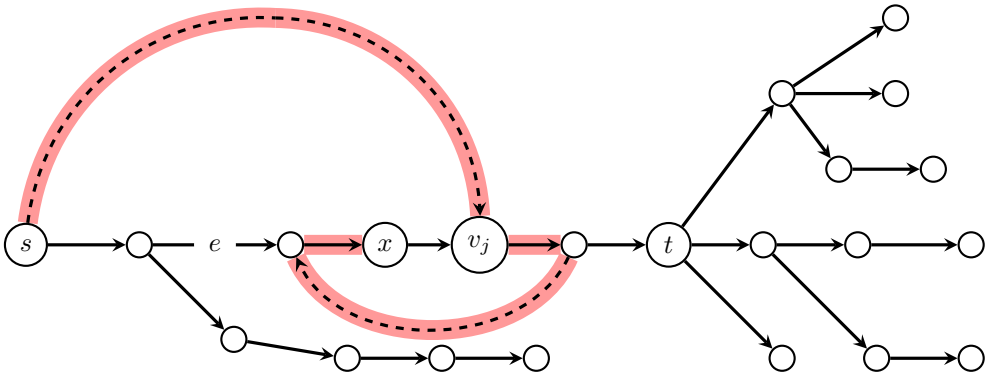

Figure 6 Case 4.2: $R(s, x, H_w - e)$ is weighted, it uses no nodes from T after v_j.

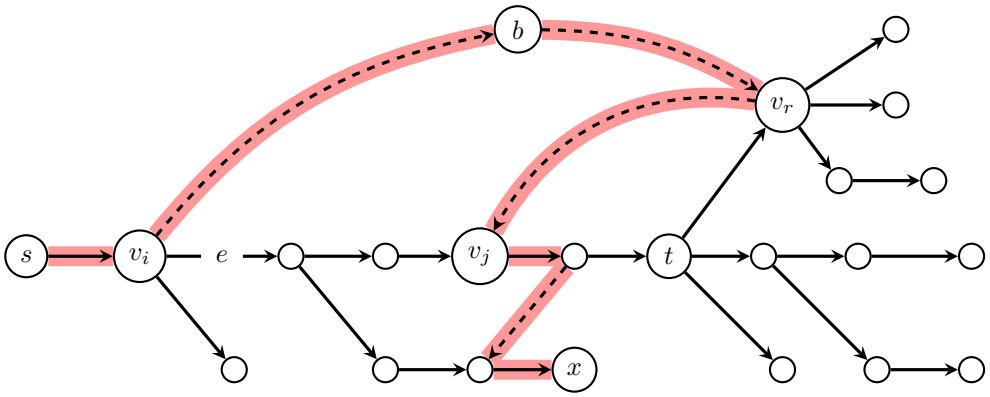

Figure 7 Case 4.3: $R(s, x, H_w - e)$ is unweighted, it uses a node from T in the sub-path from v_i to v_j, and uses no nodes from T after v_j.

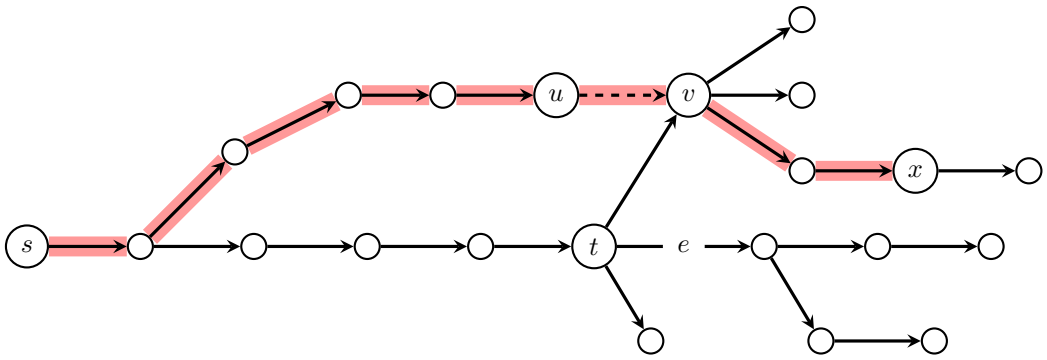

Figure 8 $R(s, x, H_w - e)$ is a "help from above" path.

References

1. Noga Alon, Zvi Galil, and Oded Margalit. On the exponent of the all pairs shortest path problem. *J. Comput. Syst. Sci.*, 54(2):255–262, April 1997. doi:10.1006/jcss.1997.1388.
2. Shiri Chechik and Sarel Cohen. Near optimal algorithms for the single source replacement paths problem. In *Proceedings of the Thirtieth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '19, pages 2090–2109, Philadelphia, PA, USA, 2019. Society for Industrial and Applied Mathematics. URL: http://dl.acm.org/citation.cfm?id=3310435.3310561.
3. Yuval Emek, David Peleg, and Liam Roditty. A near-linear time algorithm for computing replacement paths in planar directed graphs. In *Proceedings of the Nineteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '08, pages 428–435, 2008. URL: http://dl.acm.org/citation.cfm?id=1347082.1347129.
4. David Eppstein. Finding the k shortest paths. *SIAM Journal on Computing*, 28(2):652–673, 1998. doi:10.1137/S0097539795290477.
5. F. Grandoni and V. V. Williams. Improved distance sensitivity oracles via fast single-source replacement paths. In *2012 IEEE 53rd Annual Symposium on Foundations of Computer Science*, pages 748–757, October 2012. doi:10.1109/FOCS.2012.17.
6. J. Hershberger and S. Suri. Vickrey prices and shortest paths: what is an edge worth? In *Proceedings 2001 IEEE International Conference on Cluster Computing*, pages 252–259, October 2001. doi:10.1109/SFCS.2001.959899.

7 John Hershberger, Subhash Suri, and Amit Bhosle. On the difficulty of some shortest path problems. *ACM Trans. Algorithms*, 3(1):5:1–5:15, February 2007. doi:10.1145/1186810.1186815.

8 David R. Karger, Daphne Koller, and Steven J. Phillips. Finding the hidden path: Time bounds for all-pairs shortest paths. *SIAM Journal on Computing*, 22(6):1199–1217, 1993. doi:10.1137/0222071.

9 Philip N. Klein, Shay Mozes, and Oren Weimann. Shortest paths in directed planar graphs with negative lengths: A linear-space $o(n \log^2 n)$-time algorithm. *ACM Trans. Algorithms*, 6(2):30:1–30:18, April 2010. doi:10.1145/1721837.1721846.

10 François Le Gall. Powers of tensors and fast matrix multiplication. In *Proceedings of the 39th International Symposium on Symbolic and Algebraic Computation*, ISSAC '14, pages 296–303, New York, NY, USA, 2014. ACM. doi:10.1145/2608628.2608664.

11 K. Malik, A. K. Mittal, and S. K. Gupta. The k most vital arcs in the shortest path problem. *Oper. Res. Lett.*, 8(4):223–227, August 1989. doi:10.1016/0167-6377(89)90065-5.

12 Enrico Nardelli, Guido Proietti, and Peter Widmayer. A faster computation of the most vital edge of a shortest path. *Inf. Process. Lett.*, 79(2):81–85, June 2001. doi:10.1016/S0020-0190(00)00175-7.

13 Enrico Nardelli, Guido Proietti, and Peter Widmayer. Finding the most vital node of a shortest path. *Theor. Comput. Sci.*, 296(1):167–177, March 2003. doi:10.1016/S0304-3975(02)00438-3.

14 Noam Nisan and Amir Ronen. Algorithmic mechanism design (extended abstract). In *Proceedings of the Thirty-first Annual ACM Symposium on Theory of Computing*, STOC '99, pages 129–140, New York, NY, USA, 1999. ACM. doi:10.1145/301250.301287.

15 Liam Roditty and Uri Zwick. Replacement paths and k simple shortest paths in unweighted directed graphs. *ACM Trans. Algorithms*, 8(4):33:1–33:11, October 2012. doi:10.1145/2344422.2344423.

16 Virginia Vassilevska Williams. Faster replacement paths. In *Proceedings of the Twenty-Second Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '11, page 1337–1346, USA, 2011. Society for Industrial and Applied Mathematics.

17 Virginia Vassilevska Williams. Multiplying matrices faster than coppersmith-winograd. In *Proceedings of the Forty-fourth Annual ACM Symposium on Theory of Computing*, STOC '12, pages 887–898, New York, NY, USA, 2012. ACM. doi:10.1145/2213977.2214056.

18 Virginia Vassilevska Williams and R. Ryan Williams. Subcubic equivalences between path, matrix, and triangle problems. *J. ACM*, 65(5), August 2018. doi:10.1145/3186893.

19 Christian Wulff-Nilsen. Solving the replacement paths problem for planar directed graphs in $o(n \log n)$ time. In *Proceedings of the Twenty-first Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '10, pages 756–765, 2010. URL: http://dl.acm.org/citation.cfm?id=1873601.1873663.

20 Gideon Yuval. An algorithm for finding all shortest paths using $n^{2.81}$ infinite-precision multiplications. *Inf. Process. Lett.*, 4:155–156, 1976.

21 Uri Zwick. All pairs shortest paths using bridging sets and rectangular matrix multiplication. *Journal of the ACM*, 49(3):289–317, May 2002. doi:10.1145/567112.567114.