Improved Algorithms for Computing k-Sink on Dynamic Flow Path Networks

Binay Bhattacharya[1⋆], Mordecai J. Golin[2⋆⋆], Yuya Higashikawa[3,⋆⋆⋆,†], Tsunehiko Kameda[1], and Naoki Katoh[4,⋆⋆⋆]

[1] School of Computing Science, Simon Fraser University, Burnaby, Canada
[2] Dept. of Computer Science, Hong Kong Univ. of Science and Technology, China
[3] Dept. of Information and System Engineering, Chuo University, Tokyo, Japan
[4] School of Science and Technology, Kwansei Gakuin University, Hyogo, Japan

Abstract. We address the problem of locating k sinks on dynamic flow path networks with n vertices in such a way that the evacuation completion time to them is minimized. Our two algorithms run in $O(n \log n + k^2 \log^4 n)$ and $O(n \log^3 n)$ time, respectively. When all edges have the same capacity, we also present two algorithms which run in $O(n + k^2 \log^2 n)$ time and $O(n \log n)$ time, respectively. These algorithms together improve upon the previously most efficient algorithms, which have time complexities $O(kn \log^2 n)$ [1] and $O(kn)$ [11], in the general and uniform edge capacity cases, respectively. The above results are achieved by organizing relevant data for subpaths in a strategic way during preprocessing, and the final results are obtained by extracting/merging them in an efficient manner.

1 Introduction

Ford and Fulkerson [5] introduced the concept of *dynamic flow* which models movement of commodities in a network. In this model, each vertex is assigned some initial amount of supply, each edge has a capacity, which limits the rate of commodity flow into it per unit time, and the transit time to traverse it. One variant of the dynamic flow problem is the *quickest transshipment* problem, where the source vertices have specified supplies and sink vertices have specified demands. The problem is to send exactly the right amount of commodity out of sources into sinks in minimum overall time. Hoppe and Tardos [12] provided a polynomial time algorithm for this problem in the case where the transit times are integral. However, the complexity of their algorithm is very high. Finding a practical polynomial time solution to this problem is still open. The reader is referred to a recent paper by Skutella [18] on dynamic flows.

This paper discusses a related problem, called the *evacuation problem* [8, 14], in which the supplies (i.e., evacuees) are discrete, and the sinks and their demands are not specified. In fact, the locations of the sinks are the output of the problem. Many disasters, such as earthquakes, nuclear plant accidents, volcanic

⋆ Partially supported by a Discovery Grant from NSERC of Canada
⋆⋆ Partially supported by Hong Kong RGC GRF grant 16208415
⋆⋆⋆ Supported by JSPS KAKENHI Grant-in-Aid for Young Scientists (B) (17K12641)
† Supported by JST CREST (JPMJCR1402)

F. Ellen et al. (Eds.): WADS 2017, LNCS 10389, pp. 133–144, 2017.
DOI: 10.1007/978-3-319-62127-2_12

eruptions, flooding, have struck in recent years in many parts of the world, and it is recognized that orderly evacuation planning is urgently needed.

A *k-sink* is a set of k sinks such that the evacuation completion time to sinks is minimized, and our objective is to find a k-sink on a dynamic flow path network. *Congestion* is said to occur when an evacuee cannot move at the maximum speed constrained only by transit time. Thus, when the capacities of the edges are sufficiently large, no congestion occurs and each evacuee can follow the shortest path to its nearest sink at the maximum speed. This is equivalent to the classical k-center problem in networks, which is known to be NP-hard even on bipartite planar graphs of maximum degree 4 [17]. To the best of our knowledge the most general polynomially solvable case for general k is where the underlying graphs are cacti or partial t-trees with constant t. Congestion could occur if vertex capacities are limited, in which case edges may get clogged and congestion backs up. Our results are valid regardless of whether vertex capacities (the number of evacuees that they can accommodate) are limited or not.

Mamada et al. [15] solved the 1-sink problem for the dynamic flow tree networks in $O(n \log^2 n)$ time under the condition that only a vertex can be a sink, where n is the number of vertices. When edge capacities are uniform, we have presented $O(n \log n)$ time algorithms with a more relaxed condition that the sink can be on an edge, as well as on a vertex [3, 10]. Dealing with congestion is non-trivial even in path networks. On dynamic flow path networks with uniform edge capacities, it is straightforward to compute the 1-sink in linear time, as shown by Cheng et al. [4]. Arumugam et al. [1] showed that the k-sink problem for dynamic flow path networks can be solved in $O(kn \log^2 n)$ time, and when the edge capacities are uniform Higashikawa et al. [11] showed that it can be solved in $O(kn)$ time.

In this paper we present two algorithms for the k-sink problem on dynamic flow path networks with general edge capacities. A path network can model an airplane aisle, a hall way in a building, a street, a highway, etc., to name a few. Unlike the previous algorithm for the k-sink problem [1] which uses dynamic programming, our algorithms adopt Megiddo's *parametric search* [16] and the *sorted matrices* introduced by Frederickson and Johnson [6, 7]. Together, they outperform all other known algorithms, and they are the first sub-quadratic algorithms for any value of k. These improvements were made possible by our method of merging evacuation times of subpaths stored in a hierarchical data structure. We also present two algorithms for the dynamic flow path networks with uniform edge capacities.

This paper is organized as follows. In the next section, we define our model and the terms that are used throughout the paper. Sec. 3 introduces a new data structure, named the *capacities and upper envelopes tree*, which plays a central role in the rest of the paper. In Sec. 4 we identify two important tasks that form building blocks of our algorithms, and also discuss a feasibility test. Sec. 5 presents several algorithms for uniform and general edge capacities. Finally, Sec. 6 concludes the paper.

2 Preliminaries

2.1 Definitions

Let $P = (V, E)$ be a path network, whose vertices $v_1, v_2, \ldots, v_n$ are arranged from left to right in this order. For $i = 1, 2, \ldots, n$, vertex v_i has an integral weight w_i (> 0), representing the number of evacuees, and each edge $e_i = (v_i, v_{i+1})$ has a fixed non-negative length l_i and an integral *capacity* c_i, which is the upper limit on the number of evacuees who can enter an edge per unit time. We assume that a sink has infinite capacity, so that the evacuees coming from the left and right of a sink do not interfere with each other. An evacuation starts at the same time from all the vertices, and all the evacuees from a vertex evacuate to the same sink. This is called "confluent flow" in the parlance of the network flow theory. This constraint is desirable in evacuation in order to avoid confusion among the evacuees at a vertex as to which way they should move.

By $x \in P$, we mean that point x lies on either an edge or a vertex of P. For two points $a, b \in P$, $a \prec b$ or $b \succ a$ means that a lies to the left of b. Let $d(a, b)$ denote the distance (sum of the edge lengths) between a and b. If a and/or b lies on an edge, we use the prorated distance. The transit time for a unit distance is denoted by τ, so that it takes $d(a, b)\tau$ time to travel from a to b, and τ is independent of the edge. Let $c(a, b)$ denote the minimum capacity of the edges on the subpath of P between a and b. The point that is arbitrarily close to v_i on its left (resp. right) side is denoted by v_i^- (resp. v_i^+). Let $P[a, b]$ denote the subpath of P between a and b satisfying $a \prec b$. If a, b or both are excluded, we denote them by $P(a, b]$, $P[a, b)$ or $P(a, b)$, respectively. Let $V[a, b]$ (resp. $V(a, b]$, $V[a, b)$ or $V(a, b)$) denotes the set of vertices on $P[a, b]$ (resp. $P(a, b]$, $P[a, b)$ or $P(a, b)$). We introduce a weight array $W[\cdot]$, defined by

$$W[i] \triangleq \sum_{v_j \in V[v_1, v_i]} w_j, \text{ for } i = 1, 2, \ldots, n, \tag{1}$$

and let $W[v_i, v_j] \triangleq W[j] - W[i-1]$ for $i \leq j$.

2.2 Completion time functions

In our model, a set of k sinks accepts evacuees from k disjoint subpaths of P. We thus need to be able to compute the completion time for each such subpath $P[v_i, v_j]$. For simplicity, from now on, we assume that the optimal k sinks are on edges, not on vertices. Small modifications will be necessary if we allow some sinks to be on vertices. We define the *completion time from left* (*L-time* for short) to $x \succ v_j$ of vertex v_p on $P[v_i, v_j]$ to be the evacuation completion time to x for the evacuees on the vertices on $P[v_i, v_p]$, assuming that they all arrive at x continuously at a uniform rate $c(v_p, x)$. We similarly define the *completion time from right* (*R-time* for short) to $x \prec v_i$ of vertex v_p on $P[v_i, v_j]$ to be the evacuation completion time to x for all the evacuees on the vertices on $P[v_p, v_j]$, arriving at x continuously at a uniform rate $c(x, v_p)$. For any vertex $v_p \in V[v_i, v_j]$, its L-time and R-time are given mathematically as

$$\theta_L^{[i,j]}(x, v_p) \triangleq d(v_p, x)\tau + W[v_i, v_p]/c(v_p, x) \text{ for } x \succ v_j, \tag{2}$$

$$\theta_R^{[i,j]}(x, v_p) \triangleq d(x, v_p)\tau + W[v_p, v_j]/c(x, v_p) \text{ for } x \prec v_i, \tag{3}$$

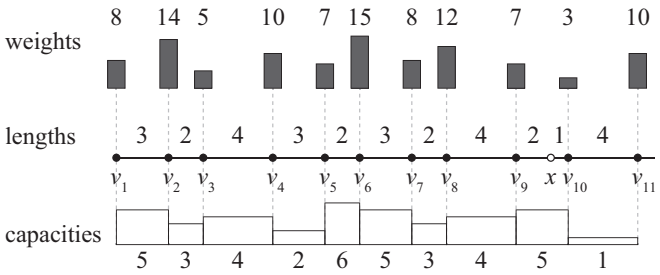

Fig. 1. An example path $P = (V, E)$ with $x \in e_9 = (v_9, v_{10})$.

respectively. For convenience we sometimes refer to the first (resp. second) term in the righthand side of (2) and (3) as the *distance time* (resp. *weight time*). Note that the distance time is linear in the distance to x. Consider an arbitrary subpath $P[v_i, v_j]$, where $i \leq j$.

Fig. 1 shows an example, where vertices $v_1, v_2, v_3, \ldots$ (represented by black circles) have weights $8, 14, 5, \ldots$, and edges $e_1, e_2, e_3, \ldots$ have lengths $3, 2, 4, \ldots$ and capacities $5, 3, 4, \ldots$ Point x is located on $e_9 = (v_9, v_{10})$ so that $d(v_9, x) = 2$ (represented by a white circle). Assuming $\tau = 1$, let us compute the L-time to x of vertex v_5 on $P[v_2, v_7]$. From $d(v_5, x) = 13$, $W[v_2, v_5] = 36$ and $c(v_5, x) = 3$, we obtain $\theta_L^{[2,7]}(x, v_5) = 25$.

To be more precise, the weight time should be $\lceil W[v_i, v_p]/c(v_p, x) \rceil$ and $\lceil W[v_p, v_j]/c(x, v_p) \rceil$ in (2) and (3), respectively, since the evacuees are discrete entities. Although only small modifications are necessary to get exact solutions as shown in [4], we use (2) and (3) for simplicity.

Lemma 1. [9] *Let s be the sink for a subpath $P[v_i, v_j]$ of a path network P. The evacuation completion time to s ($v_i \preceq v_h \prec s \prec v_{h+1} \preceq v_j$) for the evacuees on $P[v_i, v_j]$ is given by*

$$\Theta^{[i,j]}(s) \triangleq \max \left\{ \max_{v \in V[v_i, s)} \{\theta_L^{[i,h]}(s, v)\}, \max_{v' \in V(s, v_j]} \{\theta_R^{[h+1,j]}(s, v')\} \right\}. \quad (4)$$

Referring to (2) and (3), the vertex $v_p \in V[v_i, v_j]$ that maximizes $\theta_L^{[i,j]}(v_j^+, v_p)$ (resp. $\theta_R^{[i,j]}(v_i^-, v_p)$) is called the *L-critical* vertex (resp. *R-critical* vertex) of $P[v_i, v_j]$, and is denoted by $c_L^{[i,j]}$ (resp. $c_R^{[i,j]}$). Note that (v_j^+, v_p) (resp. (v_i^-, v_p)) is used instead of (v_j, v_p) (resp. (v_i, v_p)), and that we have $d(v_p, v_j^+) = d(v_p, v_j)$ and $c(v_p, v_j^+) = \min\{c(v_p, v_j), c_j\}$ (resp. $d(v_i^-, v_p) = d(v_i, v_p)$ and $c(v_i^-, v_p) = \min\{c(v_i, v_p), c_{i-1}\}$).

Using the example in Fig. 1 again, let us find the L-critical vertex of $P[v_2, v_7]$. We first compute $\theta_L^{[2,7]}(v_7^+, v_p)$ for $p = 2, \ldots, 7$: $\theta_L^{[2,7]}(v_7^+, v_2) = 14 + 14/2 = 21$, $\theta_L^{[2,7]}(v_7^+, v_3) = 12 + 19/2 = 21.5$, $\theta_L^{[2,7]}(v_7^+, v_4) = 8 + 29/2 = 22.5$, $\theta_L^{[2,7]}(v_7^+, v_5) = 5 + 36/3 = 17$, $\theta_L^{[2,7]}(v_7^+, v_6) = 3 + 51/3 = 20$, and $\theta_L^{[2,7]}(v_7^+, v_7) = 0 + 59/3 \approx 19.7$. Comparing these values, we obtain $c_L^{[2,7]} = v_4$.

Proposition 1. *Critical vertex* $v_p = c_L^{[i,j]}$ *(resp.* $v_p = c_R^{[i,j]}$*) maximizes* $\theta_L^{[i,j]}(x, v_p)$ *(resp.* $\theta_R^{[i,j]}(x, v_p)$*) for any point* $x \in (v_j, v_{j+1}]$ *(resp.* $x \in [v_{i-1}, v_i)$*).*

3 Data structures

A problem instance is said to be *t-feasible* if there are k sinks such that every evacuee can reach a sink within time t. In our algorithms, we want to perform t-feasibility tests for many different values of completion time t. Therefore, it is worthwhile to spend some time during preprocessing to construct data structures which facilitate these tests.

3.1 Capacities and upper envelopes (CUE) tree

We want to design a data structure with which critical vertices $c_L^{[i,j]}$ and $c_R^{[i,j]}$ can be found efficiently for an arbitrary pair (i, j) with $1 \le i \le j \le n$. To this end we introduce the *capacities and upper envelopes tree* (*CUE tree*, for short), denoted by $\mathcal{T}$, with root ρ, whose leaves are the vertices of P arranged from left to right. It is a balanced tree with height $O(\log n)$. In balancing, the vertex weights are not considered. For a node[5] u of $\mathcal{T}$, let $\mathcal{T}(u)$ denote the subtree rooted at u, and let $l(u)$ (resp. $r(u)$) denote the index of the leftmost (resp. rightmost) vertex on P that belongs to $\mathcal{T}(u)$. See Fig. 2. Let u_l, u_r and u_p denote the left child of u, the right child of u, and the parent of u, respectively. We say that node

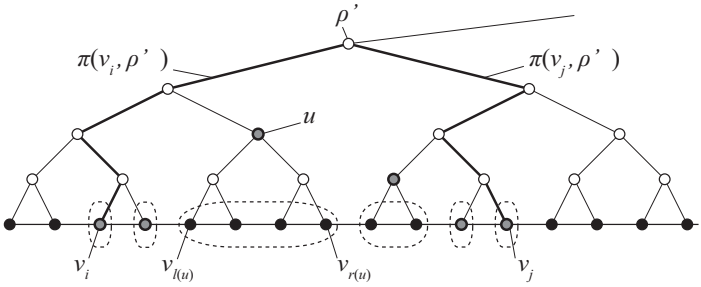

Fig. 2. Illustration of a part of CUE tree $\mathcal{T}$. The small gray disks represent nodes of $N[v_i, v_j]$ and dashed circles enclose subpaths in $\mathcal{P}[v_i, v_j]$.

u *spans* subpath $P[v_{l(u)}, v_{r(u)}]$. At node u, we store $l(u)$, $r(u)$ and the capacity $c(v_{l(u)}, v_{r(u)})$ among others. This information at every node can be computed bottom up in $O(n)$ time by performing heap-like operations.

For two nodes u, u' of $\mathcal{T}$, let $\pi(u, u')$ denote the path from u to u' along edges of $\mathcal{T}$. Suppose that for an index pair (i, j) with $1 \le i \le j \le n$, node ρ' is

[5] We use the term "node" here to distinguish it from the vertices on the path. A vertex, being a leaf of $\mathcal{T}$, is considered a node, but an interior node of $\mathcal{T}$ is not a vertex.

the lowest common ancestor of v_i and v_j in $\mathcal{T}$. Consider every node of $\mathcal{T}$ that is the right child of a node on $\pi(v_i, \rho')$ or the left child of a node on $\pi(v_j, \rho')$, but which itself is not on $\pi(v_i, \rho')$ or $\pi(v_j, \rho')$. Let $N[v_i, v_j]$ denote the set of such nodes plus v_i and v_j. Then clearly $N[v_i, v_j]$ consists of $O(\log n)$ nodes. Let $\mathcal{P}[v_i, v_j]$ denote the set of $O(\log n)$ subpaths spanned by nodes of $N[v_i, v_j]$.

In order to determine $c_L^{[i,j]}$ for a given pair (i, j), we need to compute

$$\max_{v_p \in V[v_i, v_j]} \{d(v_p, v_j)\tau + W[v_i, v_p]/c(v_p, v_{j+1})\}. \tag{5}$$

To facilitate such a computation for an arbitrary pair (i, j), at each node u, we precompute and store two upper envelope functions associated with subpath $P[v_{l(u)}, v_{r(u)}]$. Then for $u \in N[v_i, v_j]$ that spans v_p, we have $W[v_i, v_p] = W[v_i, v_{l(u)-1}] + W[v_{l(u)}, v_p]$ and $c(v_p, v_{j+1}) = \min\{c(v_p, v_{r(u)+1}), c(v_{r(u)+1}, v_{j+1})\}$. Since (i, j), hence $W[v_i, v_{l(u)-1}]$ and $c(v_{r(u)+1}, v_{j+1})$, is not known during preprocessing, we replace these values with variables W and c, respectively, and express the two upper envelopes stored at u as functions of $W = W[v_i, v_{l(u)-1}]$ and $c = c(v_{r(u)+1}, v_{j+1})$, respectively. We can now break (5) down into a number of formulea, one for each $u \in N[v_i, v_j]$, which is given by

$$\max_{v_p \in V[v_{l(u)}, v_{r(u)}]} \left\{d(v_p, v_{r(u)})\tau + (W + W[v_{l(u)}, v_p])/\min\{c(v_p, v_{r(u)+1}), c\}\right\}. \tag{6}$$

Using the concrete values of W and c, we can evaluate (5) by finding the maximum of the $|N[v_i, v_j]| = O(\log n)$ values, computed by (6).

Now we want to compute (6) efficiently for "arbitrary" W and c, but of course we have $W = W[v_i, v_{l(u)-1}]$ and $c = c(v_{r(u)+1}, v_j)$ in mind for some i and j. Consider two extreme cases, where for any p with $v_p \in V[v_{l(u)}, v_{r(u)}]$ (i) $c > c(v_p, v_{r(u)+1})$, and (ii) $c \leq c(v_p, v_{r(u)+1})$, respectively. In Case (i), we have from (6)

$$\Theta_{L,1}^u(W) \triangleq \max_{v_p \in V[v_{l(u)}, v_{r(u)}]} \left\{d(v_p, v_{r(u)})\tau + (W + W[v_{l(u)}, v_p])/c(v_p, v_{r(u)+1})\right\} \tag{7}$$

$$= \max_{v_p \in V[v_{l(u)}, v_{r(u)}]} \left\{\theta_L^{[l(u), r(u)]}(v_{r(u)}^+, v_p) + W/c(v_p, v_{r(u)+1})\right\}. \tag{8}$$

Note that $c(v_p, v_{r(u)+1})$ gets smaller as v_p moves to the left. From (7) it is seen that $\Theta_{L,1}^u(W)$ is the upper envelope of linear functions of W and each coefficient of W is positive, which means that $\Theta_{L,1}^u(W)$ is piecewise linear, continuous, and increasing in W. Thus it can be encoded as a sequence of bending points. In Case (ii), we have from (6)

$$\Theta_{L,2}^u(c) = \max_{v_p \in V[v_{l(u)}, v_{r(u)}]} \left\{d(v_p, v_{r(u)})\tau + W[v_{l(u)}, v_p]/c\right\}. \tag{9}$$

Note that (9) was obtained from (6) by removing the term W/c, which does not depend on v_p. If we plot $\Theta_{L,2}^u(c)$ vs. $(1/c)$ as a graph, it is also piecewise linear, continuous, and increasing in $(1/c)$, and can be encoded as a sequence of bending points.

At node u we store both $\Theta^u_{L,1}(W)$ and $\Theta^u_{L,2}(c)$ in encoded form with bending points, which can be computed in $O(r(u) - l(u))$ time. Similarly, in order to determine $c_R^{[i,j]}$ for an arbitrary pair (i, j), we store two functions which are symmetric to $\Theta^u_{L,1}(W)$ and $\Theta^u_{L,2}(c)$, respectively, named $\Theta^u_{R,1}(W)$ and $\Theta^u_{R,2}(c)$, in linear time. We can now prove the following lemma.

Lemma 2. *Given a dynamic flow path network with n vertices, CUE tree $\mathcal{T}$ with associated data can be constructed in $O(n \log n)$ time and $O(n \log n)$ space.*

3.2 Using CUE tree

Suppose we want to find the L-critical vertex $c_L^{[i,j]}$ for subpath $P[v_i, v_j]$. We work on $P[v_{l(u)}, v_{r(u)}] \in \mathcal{P}[v_i, v_j]$ for each node $u \in N[v_i, v_j]$. Each such subpath provides a *candidate* for $c_L^{[i,j]}$. Clearly, among those candidates, the one that has the largest L-time is $c_L^{[i,j]}$.

Let us first compute $c = c(v_{r(u)+1}, v_{j+1})$. For this purpose, we trace the path $\pi(v_{r(u)+1}, v_{j+1})$ in $\mathcal{T}$ and, at each node $u' \in N[v_{r(u)+1}, v_{j+1}] \setminus \{v_{j+1}\}$, retrieve $c(v_{l(u')}, v_{r(u')})$ and $c_{r(u')+1}$. Taking the minimum of the retrieved capacities, we obtain $c(v_{r(u)+1}, v_{j+1})$, which costs $O(\log n)$ time.

Using binary search, we then find the largest index q ($l(u) \le q \le r(u)$), if any, such that $c(v_q, v_{r(u)+1}) < c = c(v_{r(u)+1}, v_{j+1})$ holds. Note that $c(v_q, v_{r(u)+1})$ is monotonically non-increasing as q decreases. To find q we trace the path $\pi(v_{r(u)}, u)$ in $\mathcal{T}$ as follows. Set $c_{\min}$ to $c_{r(u)+1}$ and u' to $v_{r(u)}$. If $c_{\min} < c$, q is determined as $r(u)$. Otherwise update u' to u'_p. While $u' \neq u$ and

$$\min\{c_{\min}, c(v_{l(u'_l)}, v_{r(u'_l)}), c_{r(u'_l)+1}\} \ge c, \tag{10}$$

update $c_{\min}$ to the L.H.S. of (10), and u' to u'_p. If $u' = u$ and (10) holds, such q does not exist. If (10) stops holding at some node u', then update u' to u'_l. While

$$\min\{c_{\min}, c(v_{l(u'_r)}, v_{r(u'_r)}), c_{r(u'_r)+1}\} \ge c, \tag{11}$$

update $c_{\min}$ to the L.H.S. of (11) and u' to u'_r. If (11) stops holding at some node u', then update u' to u'_r. This way we will eventually reach v_q, if it exists, in $O(\log n)$ time. If q exists, we partition $P[v_{l(u)}, v_{r(u)}]$ into two subpaths $P[v_{l(u)}, v_q]$ and $P[v_{q+1}, v_{r(u)}]$. Letting $V_1 = V[v_{l(u)}, v_q]$, $V_2 = V[v_{q+1}, v_{r(u)}]$, and $W = W[v_i, v_{l(u)-1}]$, we define

$$\tilde{\Theta}^u_{L,1}(W) = \max_{v_p \in V_1} \left\{ \theta_L^{[l(u),r(u)]}(v^+_{r(u)}, v_p) + W/c(v_p, v_{r(u)+1}) \right\}, \tag{12}$$

$$\tilde{\Theta}^u_{L,2}(c) = \max_{v_p \in V_2} \left\{ d(v_p, v_{r(u)})\tau + W[v_{l(u)}, v_p]/c \right\}. \tag{13}$$

Note that the range of maximization $v_p \in V_1$ in (12) (resp. $v_p \in V_2$ in (13)) is limited compared with (8) (resp. (9)). If q does not exist, we set $\tilde{\Theta}^u_{L,1}(W) = 0$ and $V_2 = V[v_{l(u)}, v_{r(u)}]$. It is clear that

$$\max \left\{ \tilde{\Theta}^u_{L,1}(W), \tilde{\Theta}^u_{L,2}(c) + W/c \right\} \tag{14}$$

is equal to (6), and its maximizing vertex corresponds to a candidate from $P[v_{l(u)}, v_{r(u)}]$ for the L-critical vertex of $P[v_i, v_j]$.

Let v_1^* (resp. v_2^*) be a vertex in $V_1 \cup V_2 = V[v_{l(u)}, v_{r(u)}]$ which maximizes the bracketed term in (8) (resp. (9)). Once $W = W[v_i, v_{l(u)-1}]$ and $c = c(v_{r(u)+1}, v_{j+1})$ are given, we can obtain v_1^* and v_2^* by binary search on the bending points of $\Theta^u_{L,1}(W)$ and $\Theta^u_{L,2}(c)$, respectively, which can be done in $O(\log n)$ time. We can now prove the following lemma.

Lemma 3.

(a) If $v_2^* \in V_1$, we have

$$\tilde{\Theta}^u_{L,1}(W) > \tilde{\Theta}^u_{L,2}(c) + W/c. \tag{15}$$

(b) If $v_1^* \in V_2$, we have

$$\tilde{\Theta}^u_{L,1}(W) \leq \tilde{\Theta}^u_{L,2}(c) + W/c. \tag{16}$$

(c) $v_1^* \in V_2$ and $v_2^* \in V_1$ cannot happen at the same time.

If $v_1^* \in V_1$ and $v_2^* \in V_2$, clearly v_1^* and v_2^* also achieve the maxima in (12) and (13), respectively. Therefore, if $\tilde{\Theta}^u_{L,1}(W) > \tilde{\Theta}^u_{L,2}(c) + W/c$ (resp. $\tilde{\Theta}^u_{L,1}(W) \leq \tilde{\Theta}^u_{L,2}(c) + W/c$), v_1^* (resp. v_2^*) is a candidate critical vertex from $P[v_{l(u)}, v_{r(u)}]$. Otherwise, by Lemma 3, $v_1^*, v_2^* \in V_1$ or $v_1^*, v_2^* \in V_2$ holds. Also by Lemma 3, if $v_1^*, v_2^* \in V_1$ (resp. $v_1^*, v_2^* \in V_2$), v_1^* (resp. v_2^*) is a candidate critical vertex. Based on the above arguments, we can prove the following lemma.

Lemma 4. *Suppose that CUE tree $\mathcal{T}$ is available. Consider subpath $P[v_i, v_j]$ with $1 \leq i < j \leq n$.*

(a) For each node $u \in N[v_i, v_j]$, candidates from $P[v_{l(u)}, v_{r(u)}]$ for L-critical and R-critical vertices of $P[v_i, v_j]$ can be computed in $O(\log n)$ time.
(b) The L-critical and R-critical vertices for $P[v_i, v_j]$ can be computed in $O(\log^2 n)$ time.

4 Building blocks

There are two useful tasks that we can call upon repeatedly. Given the starting vertex v_a, the first task is to find the rightmost vertex v_d such that all the evacuees on $V[v_a, v_d]$ can evacuate to a sink within time t. The second task is to find the cost of the 1-sink on a given subpath $P[v_i, v_j]$. To perform these tasks, we start with more basic procedures.

4.1 Basic algorithms

To implement the first task, note that for a given index $h > a$, there are $O(\log n)$ nodes in $N[v_a, v_h]$. For each such node u, we want to test where a sink s should be placed: to the left of $v_{l(u)}$, to the right of $v_{r(u)}$, or between $v_{l(u)}$ and $v_{r(u)}$.

Here is an algorithm for the first task.

Algorithm 1 `1-Sink`(t, v_a)

1. *Compute an integer b by binary search over h with $a \leq h \leq n$ such that the L-time of $c_L^{[a,b]}$ to v_b^+ does not exceed t but the L-time of $c_L^{[a,b+1]}$ to v_{b+1}^+ exceeds t.*
2. *Solve $\theta_L^{[a,b]}(v_b^+, c_L^{[a,b]}) + x\tau = t$, and place a sink $s \in (v_b, v_{b+1}]$ satisfying $d(v_b, s) = x$.*
3. *If $s \in (v_b, v_{b+1})$, set c to $b+1$. If $s = v_{b+1}$, set c to $b+2$. Compute an integer d by binary search over h with $c \leq h \leq n$ such that the R-time of $c_R^{[c,d]}$ to s does not exceed t but the R-time of $c_R^{[c,d+1]}$ to s exceeds t.*

Lemma 5. *If CUE tree $\mathcal{T}$ is available,* `1-Sink`(t, v_a) *runs in $O(\log^3 n)$ time.*

Proof. In Step 1, for a fixed h, finding $c_L^{[a,h]}$ and computing the L-time of $c_L^{[a,h]}$ to v_h^+ take $O(\log^2 n)$ time by Lemma 4. Clearly, we repeat this computation $O(\log n)$ times, thus Step 1 takes $O(\log^3 n)$ time. Step 2 takes $O(1)$ time and Step 3 takes $O(\log^3 n)$ time similarly to Step 1. Summarizing these, we complete the proof. □

Here is an algorithm for the second task.

Algorithm 2 `Local-Cost`(v_i, v_j)

1. *Let u be the node where the two paths $\pi(v_i, \rho)$ and $\pi(v_j, \rho)$ meet.*
2. *If the L-time of $c_L^{[i,r(u_l)]}$ and the R-time of $c_R^{[l(u_r),j]}$ have the same value at some point x on the edge $(v_{r(u_l)}, v_{l(u_r)})$, then output x as the 1-sink.*
3. *If the L-time of $c_L^{[i,r(u_l)]}$ is higher (resp. lower) than the R-time of $c_R^{[l(u_r),j]}$ at every point on edge $(v_{r(u_l)}, v_{l(u_r)})$, then let $u = u_l$ (resp. $u = u_r$) and repeat Step 2, using the new u_l and u_r.*

We have the following lemma.

Lemma 6. *If CUE tree $\mathcal{T}$ is available,* `Local-Cost`(v_i, v_j) *finds a 1-sink on subpath $P[v_i, v_j]$ in $O(\log^3 n)$ time.*

4.2 t-feasibility test

We carry out `1-Sink`(t, v) repeatedly, starting from the left end of P, i.e., v_1. Clearly, the problem instance is t-feasible if and only if the rightmost vertex v_n belongs to the l-th isolated subpath, where $l \leq k$.

Lemma 7. *Given a dynamic flow path network, if CUE tree $\mathcal{T}$ is available, we can test its t-feasibility in $O(\min\{n \log^2 n, k \log^3 n\})$ time.*

Proof. Starting at the leftmost vertex v_1 of P, invoke `1-Sink`(t, v_1), which isolates the first subpath in $O(\log^3 n)$ time by Lemma 5, and remove it from P. We repeat this at most $k-1$ more times on the remaining subpath, spending $O(k \log^3 n)$ time.

On the other hand, when each `1-Sink`(t, v_a) is executed, suppose we compute the L-time of $c_L^{[a,h]}$ to v_h^+ for $h = a, a+1, \ldots$ one by one at Step 1, and similarly

the R-time of $c_R^{[c,h]}$ to s for $h = c, c+1, \ldots$ one by one, instead of binary search. Then, the computations of L-time and R-time are invoked at most n times during a t-feasibility test. Since each computation of L-time or R-time takes $O(\log^2 n)$ time by Lemma 4, the total time is $O(n \log^2 n)$ in this way. □

4.3 Uniform edge capacity case

The problem is much simplified if the edges have the same capacity. In particular, we can compute the critical vertex of a subpath resulting from concatenating two subpaths in constant time. At each node u of $\mathcal{T}$ bottom up, we compute and record the L- and R-critical vertices of $P[v_{l(u)}, v_{r(u)}]$ with respect to $v^+_{r(u)}$ and their costs, based on the following lemma.

Lemma 8. [11] *For a node u of CUE tree $\mathcal{T}$, let $v_{l(u_l)} = v_h$, $v_{r(u_l)} = v_i$, $v_{l(u_r)} = v_{i+1}$, and $v_{r(u_r)} = v_j$, and assume that the critical vertices, $c_L^{[h,j]}$, $c_R^{[h,j]}$, $c_L^{[i+1,j]}$, and $c_R^{[i+1,j]}$ have already been computed.*

(a) The L-critical vertex $c_L^{[h,j]}$ is either $c_L^{[h,i]}$ or $c_L^{[i+1,j]}$.
(b) The R-critical vertex $c_R^{[h,j]}$ is either $c_R^{[h,i]}$ or $c_R^{[i+1,j]}$.

The following two lemmas provide counterparts to Lemmas 2 and 4, respectively.

Lemma 9. *Given a dynamic flow path network with n vertices and uniform edge capacities, CUE tree $\mathcal{T}$ with associated data can be constructed in $O(n)$ time and $O(n)$ space.*

Lemma 10. *Suppose that CUE tree $\mathcal{T}$ is available. For any i and j $(1 \le i < j \le n)$, we can comput the L-critical and R-critical vertices for $P[v_i, v_j]$ in $O(\log n)$ time.*

Similarly to Lemma 7, we can prove the following lemma.

Lemma 11. *Given a dynamic flow path network with uniform edge capacities, if CUE tree $\mathcal{T}$ is available, we can test its t-feasibility in $O(\min\{n, k \log n\})$ time.*

5 Optimization

5.1 Parametric search approach

Lemma 12. [1] *If t-feasibility test can be tested in $\alpha(t)$ time, then the k-sink can be found in $O(k\alpha(t) \log n)$ time, excluding the preprocessing time.*

By Lemma 2 it takes $O(n \log n)$ time to construct $\mathcal{T}$ with weight and capacity data, and $\alpha(t) = O(k \log^3 n)$ by Lemma 7. Lemma 12 thus implies

Theorem 1. *Given a dynamic flow path network with n vertices, we can find an optimal k-sink in $O(n \log n + k^2 \log^4 n)$ time.*

Applying Megiddo's main theorem in [16] to Lemma 11, we obtain

Theorem 2. *Given a dynamic flow path network with n vertices and uniform edge capacities, we can find an optimal k-sink in $O(n + k^2 \log^2 n)$ time.*

5.2 Sorted matrix approach

Let $OPT(l, r)$ denote the evacuation time for the optimal 1-sink on subpath $P[v_l, v_r]$. Define an $n \times n$ matrix A whose entry (i, j) entry is given by

$$A[i,j] = \begin{cases} OPT(n-i+1, j) & \text{if } n-i+1 \leq j \\ 0 & \text{otherwise.} \end{cases} \tag{17}$$

It is clear that matrix A includes $OPT(l, r)$ for every pair of integers (l, r) such that $1 \leq l \leq r \leq n$. There exists a pair (l, r) such that $OPT(l, r)$ is the evacuation time for the optimal k-sink on the whole path. Then the k-sink location problem can be formulated as: "Find the smallest $A[i, j]$ such that the given problem instance is $A[i, j]$-feasible." Note that we do not actually compute all the elements of $A[\,]$, but element $A[i, j]$ is computed on demand as needed.

A matrix is called a *sorted matrix* if each row and column of it is sorted in the nondecreasing order. In [6, 7], Frederickson and Johnson show how to search for an element in a sorted matrix. The following lemma is implicit in their papers.

Lemma 13. *Suppose that $A[i, j]$ can be computed in $g(n)$ time, and feasibility can be tested in $f(n)$ time with $h(n)$ preprocessing time. Then we can solve the k-sink problem in $O(h(n) + ng(n) + f(n) \log n)$ time.*

We have $h(n) = O(n \log n)$ by Lemma 2, $g(n) = O(\log^3 n)$ by Lemma 6, and $f(n) = O(n \log^2 n)$ by Lemma 7. Lemma 13 thus implies

Theorem 3. *Given a dynamic path network with n vertices and general edge capacities, we can find an optimal k-sink in $O(n \log^3 n)$ time.*

In the uniform edge capacity case, we have $h(n) = O(n)$ by Lemma 9, $g(n) = O(\log n)$ by Lemma 10, and $f(n) = O(n)$ by Lemma 11. Lemma 13 thus implies

Theorem 4. *Given a dynamic path network with n vertices and uniform edge capacities, we can find the k-sink in $O(n \log n)$ time.*

6 Conclusion and discussion

We have presented more efficient algorithms than the existing ones to solve the k-sink problem on dynamic flow path networks. Due to lack of space, we could not present all the proofs. All our results are valid if the model is changed slightly, so that the weights and edge capacities are not restricted to be integers. Then it becomes confluent transshipment problem.

For dynamic flow tree networks with uniform edge capacities, it is known that computing evacuation time to a vertex can be transformed to that on a path network [13]. We believe that our method is applicable to each "spine," which is a path in the spine decomposition of a tree [2], and we think we may be able to solve the k-sink problem on dynamic flow tree networks more efficiently. This is work in progress.

References

1. G. P. Arumugam, J. Augustine, M. J. Golin, and P. Srikanthan. A polynomial time algorithm for minimax-regret evacuation on a dynamic path. *arXiv:1404,5448v1*, 2014.
2. R. Benkoczi, B. Bhattacharya, M. Chrobak, L. Larmore, and W. Rytter. Faster algorithms for k-median problems in trees. *Mathematical Foundations of Computer Science, Springer-Verlag*, LNCS 2747:218–227, 2003.
3. Binay Bhattacharya and Tsunehiko Kameda. Improved algorithms for computing minmax regret sinks on path and tree networks. *Theoretical Computer Science*, 607:411–425, Nov. 2015.
4. S. W. Cheng, Y. Higashikawa, N. Katoh, G. Ni, B. Su, and Y. Xu. Minimax regret 1-sink location problem in dynamic path networks. In *Proc. Annual Conf. on Theory and Applications of Models of Computation (T-H.H. Chan, L.C. Lau, and L. Trevisan, Eds.), Springer-Verlag*, volume LNCS 7876, pages 121–132, 2013.
5. L. R. Ford and D. R. Fulkerson. Constructing maximal dynamic flows from static flows. *Operations research*, 6(3):419–433, 1958.
6. G. N. Frederickson. Optimal algorithms for tree partitioning. In *Proc. 2nd ACM-SIAM Symp. Discrete Algorithms*, pages 168–177, 1991.
7. G. N. Frederickson and D. B. Johnson. Finding kth paths and p-centers by generating and searching good data structures. *J. Algorithms*, 4:61–80, 1983.
8. H.W. Hamacher and S.A. Tjandra. Mathematical modeling of evacuation problems: a state of the art. *in: Pedestrian and Evacuation Dynamics, Springer Verlag,*, pages 227–266, 2002.
9. Y. Higashikawa. *Studies on the space exploration and the sink location under incomplete information towards applications to evacuation planning.* PhD thesis, Kyoto University, Japan, 2014.
10. Y. Higashikawa, M. J. Golin, and N. Katoh. Minimax regret sink location problem in dynamic tree networks with uniform capacity. *J. of Graph Algorithms and Applications*, 18.4:539–555, 2014.
11. Y. Higashikawa, M. J. Golin, and N. Katoh. Multiple sink location problems in dynamic path networks. *Theoretical Computer Science*, 607:2–15, 2015.
12. B. Hoppe and É. Tardos. The quickest transshipment problem. *Mathematics of Operations Research*, 25(1):36–62, 2000.
13. N. Kamiyama, N. Katoh, and A. Takizawa. An efficient algorithm for evacuation problem in dynamic network flows with uniform arc capacity. *IEICE Transactions*, 89-D(8):2372–2379, 2006.
14. S. Mamada, K. Makino, and S. Fujishige. Optimal sink location problem for dynamic flows in a tree network. *IEICE Trans. Fundamentals*, E85-A:1020–1025, 2002.
15. S. Mamada, T. Uno, K. Makino, and S. Fujishige. An $O(n \log^2 n)$ algorithm for a sink location problem in dynamic tree networks. *Discrete Applied Mathematics*, 154:2387–2401, 2006.
16. N. Megiddo. Combinatorial optimization with rational objective functions. *Math. Oper. Res.*, 4:414–424, 1979.
17. N. Megiddo and A. Tamir. New results on the complexity of p-center problems. *SIAM J. Comput.*, 12:751–758, 1983.
18. M. Skutella. An introduction to network flows over time. In *Research Trends in Combinatorial Optimization*, pages 451–482. Springer, 2009.