Minsum k-Sink Problem on Dynamic Flow Path Networks

Robert Benkoczi[1], Binay Bhattacharya[2], Yuya Higashikawa[3], Tsunehiko Kameda[2(✉)], and Naoki Katoh[4]

[1] Department of Mathematics and Computer Science, University of Lethbridge, Lethbridge, Canada
[2] School of Computing Science, Simon Fraser University, Burnaby, Canada
tiko@sfu.ca
[3] School of Business Administration, University of Hyogo, Kobe, Japan
[4] School of Science and Technology, Kwansei Gakuin University, Sanda, Japan

Abstract. In emergencies such as earthquakes, nuclear accidents, etc., we need an evacuation plan. We model a street, a building corridor, etc. by a path network, and consider the problem of locating a set of k sinks on a dynamic flow path network with n vertices, where people are located, that minimizes the sum of the evacuation times of all evacuees. Our minsum model is more difficult to deal with than the minmax model, because the cost function is not monotone along the path. We present an $O(kn^2 \log^2 n)$ time algorithm for solving this problem, which is the first polynomial time result. If the edge capacities are uniform, we give an $O(kn \log^3 n)$ time algorithm.

1 Introduction

Due to many recent disasters such as earthquakes, volcanic eruptions, hurricanes, and nuclear plant accidents, evacuation planning is getting increasing attention. The evacuation k-sink problem is an attempt to model evacuation in such emergency situations [5,6]. In this paper, a *k-sink* means a set of k sinks that minimizes the sum of the evacuation time of every evacuee to a sink. Researchers have worked mainly on two objective functions. One is the evacuation completion time (minmax criterion), and the other is the sum of the evacuation times of all the evacuees (minsum criterion). It is assumed that all evacuees from a vertex evacuate to the same sink.

Mamada et al. [12] solved the minmax 1-sink problem for dynamic flow tree networks in $O(n \log^2 n)$ time under the condition that only a vertex can be a sink. When edge capacities are uniform, Higashikawa et al. [9] and Bhattacharya and Kameda [3] presented $O(n \log n)$ time algorithms with a more relaxed

This work was supported in part by NSERC Discovery Grants, awarded to R. Benkoczi and B. Bhattacharya, in part by JST CREST Grant Number JPMJCR1402 held by N. Katoh and Y. Higashikawa, and in part by JSPS Kakenhi Grant-in-Aid for Young Scientists (B) (17K12641) given to Y. Higashikawa.

C. Iliopoulos et al. (Eds.): IWOCA 2018, LNCS 10979, pp. 78–89, 2018.
https://doi.org/10.1007/978-3-319-94667-2_7

condition that the sink can be on an edge. Chen and Golin [4] solved the minmax k-sink problem on dynamic flow tree networks in $O(k^2 n \log^5 n)$ time when the edge capacities are non-uniform. Regarding the minmax k-sink on dynamic flow path networks, Higashikawa et al. [10] present an algorithm to compute a k-sink in $O(kn)$ time if the edge capacities are uniform. In the general edge capacity case, Arumugam et al. [1] showed that a k-sink can be found in $O(kn \log^2 n)$ time. Bhattacharya et al. [2] recently improved these results to $O(\min\{n \log n, n + k^2 \log^2 n)$ time in the uniform edge capacity case, and to $O(\min\{n \log^3 n, n \log n + k^2 \log^4 n\})$ time in the general case.

The minsum objective function for the sink problems is motivated, among others, by the desire to minimize the transportation cost of evacuation or the total amount of psychological duress suffered by the evacuees. It is more difficult than the minmax variety because the objective cost function is not unimodal, and, to the best of our knowledge, practically nothing is known about this problem on more general networks than path networks. A path network, although simple, can model an airplane aisle, a hall way in a building, a street, a highway, etc., to name a few. For the simplest case of $k = 1$ and uniform edge capacities, Higashikawa et al. [10] proposed an $O(n)$ time algorithm. For the case of general k and uniform edge capacities, Higashikawa et al. [10] showed that a k-sink can be found in time bounded by $O(kn^2)$ and $2^{O(\sqrt{\log k \log\log n})} n^2$.

The main contribution of this paper is an $O(kn^2 \log^2 n)$ time algorithm, which is achieved by a novel data structure and the concepts of *cluster* and *section* in an evacuee flow. Our second algorithm solves the problem in $O(kn \log^3 n)$ time if the edge capacities are the same.

This paper is organized as follows. In the next section, we define some terms that are used throughout this paper, and present a few basic facts. Section 3 formulates the framework for solving the minsum k-sink problem, utilizing Dynamic Programming (DP), and provides a solution. In Sect. 4, we introduce the concepts of cluster and section which play a key role in subsequent discussions, and discuss how to compute the local cost data that are required in our DP formulation. Section 5 states our main theorem, which results from the preceding section. Finally, Sect. 6 concludes the paper.

2 Preliminaries

Let $P(V, E)$ denote a given path network, where the vertex set V consists of $v_1, v_2, \ldots, v_n$, which we assume to be arranged in this order, from left to right horizontally.[1] Vertex v_i has weight $w_i \in \mathbb{Z}_+$, representing the number of evacuees initially located at v_i, and edge $e_i = (v_i, v_{i+1}) \in E$ has *length* or *distance* d_i (> 0) and *capacity* c_i, which is the upper limit on the flow rate through e_i in persons/unit time. We write $v_i \prec v_j$ if $i < j$. For two vertices $v_i \prec v_j$, the subpath between them is denoted by $P[v_i, v_j]$, and $d(v_i, v_j)$ (resp. $c(v_i, v_j)$) denotes

[1] In Sects. 3 and 4, for simplicity, we will often identify a vertex with its index, referring to vertex i, instead of vertex v_i.

its length (resp. the minimum capacity of the edges on $P[v_i, v_j]$). It takes each evacuee τ units of time to travel a unit distance.

Our model assumes that the evacuees at all the vertices start evacuation at the same time at the rate limited by the capacity of the outgoing edge. It also assumes that all the evacuees at a non-sink vertex who were initially there or who arrive there later evacuate in the same direction (either to the left or to the right), i.e., the evacuee flow is *confluent*. We sometimes use the term "cost" to refer to the aggregate evacuation time of a group of evacuees to a certain destination. A *k-sink* shares the following property of the *median* problem [11].

Lemma 1 [10]**.** *There is a k-sink such that all the k sinks are at vertices.*

If we plot the arrival flow rate at, or departure flow rate from, a vertex as a function of time, it consists of a sequence of *(temporal) clusters*. The *duration* of a cluster is the length of time in which the flow rate corresponding to the cluster is greater than zero. A cluster consists of a sequence of *sections*, such that any adjacent pair of sections have different heights. In other words, a section is a maximal part of a cluster with the same *height* (= flow rate). A *simple cluster* consists of just one section. We say that a cluster/section *carries* (the evacuees on) a vertex, if those evacuees provide flow to the cluster/section. A time interval of flow rate 0 between adjacent clusters is called a *gap*. These terms are illustrated in Fig. 1. Unless otherwise specified, we assume that evacuees arrive at vertex v_i from vertex v_{i+1}. The case where the evacuees move rightward can be treated symmetrically.

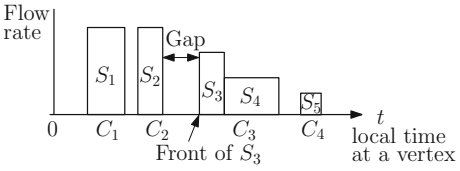

Fig. 1. Terms used: $\{S_i\}$ are sections and $\{C_j\}$ are clusters.

The *front* of a cluster/section is the time when it starts. The *first vertex* of a cluster is the vertex from which the evacuee corresponding to the front of the cluster originates. The *offset* of a cluster with respect to vertex v_i is the time until the first evacuee belonging to the cluster arrives at v_i. For $v_j \prec x \prec v_{j+1}$, we define the following costs.

$$\Phi_{L,j}(x) \triangleq \text{cost contribution to } x \text{ from } P[v_1, v_j],$$
$$\Phi_{R,j}(x) \triangleq \text{cost contribution to } x \text{ from } P[v_{j+1}, v_n],$$
$$\Phi(x) = \begin{cases} \Phi_{L,j}(x) + \Phi_{R,j+1}(x) & \text{if } v_j \prec x \prec v_{j+1} \\ \Phi_{L,j-1}(x) + \Phi_{R,j+1}(x) & \text{if } x = v_j. \end{cases} \quad (1)$$

A point $x = \mu$ that minimizes $\Phi(x)$ is called a *minsum 1-sink*.

The total cost is the sum of the costs of all sections. The cost of a section of height c with offset t_0 and duration δ_t is given by

$$\lambda t_0 + \frac{\lambda^2}{2c}, \tag{2}$$

where $\lambda = c\delta_t$ is the number of evacuees carried by the section [8]. To be exact, the ceiling function must be applied to the second term in (2), but we omit it for simplicity, and *adopt* (2) as our objective function [5]. Or we can consider each molecule of a fluid-like material as an "evacuee." The average evacuation time for an evacuee carried by this section is $t_0 + \lambda/2c$, where $\lambda/2c$ represents the average delay to reach the front vertex of the section, and the aggregate is given by $(t_0 + \lambda/2c) \times \lambda$, which yields (2). We call the first (resp. second) term in (2) the *extra cost* (resp. *intra cost*) of the section. A *minsum k-sink* partitions the path into k subpaths, and places a 1-sink on each subpath in such a way that the sum of the evacuation time of every evacuee to a sink is minimized.

3 DP Formulation

We first present a dynamic programming (DP) formulation that follows the template of recursive functions proposed by Hassin and Tamir [7] for the p-median problem. Our innovation consists in the manner in which we process the recursive computations efficiently, given that the cost functions for the sink location problem are significantly more difficult to compute than those for the regular median problem. Our algorithm is more general in that it relies only on one fundamental property of some cost functions, i.e., monotonicity.

3.1 Derivation of Recurrence Formulae

Let $F^k(i)$, $1 \le k \le i \le n$, denote the minsum cost when k sinks are placed on subpath $P[v_1, v_i]$. Similarly, define $G^k(i)$, $1 \le k \le i \le n$, as the minsum cost when k sinks are placed on subpath $P[v_1, v_i]$, *and* v_i is the rightmost sink. We start with $i = k+1$, since $F^k(i) = G^k(i) = 0$ for $i \le k$. For $j < i$, we also define $R(j, i)$, which is the cost of evacuating all the evacuees on subpath $P[v_{j+1}, v_i]$ to v_j, and $L(j, i)$, which is the cost of evacuating all the evacuees on subpath $P[v_j, v_{i-1}]$ to v_i. By definition, we have

$$F^k(i) = \min_{k \le j \le i} \{G^k(j) + R(j, i)\}, \tag{3}$$

$$G^k(i) = \min_{k \le j \le i} \{F^{k-1}(j) + L(j+1, i)\}. \tag{4}$$

To solve the above recursive equations, we clearly need to compute functions $R(j, i)$ and $L(j, i)$. Moreover, to obtain a DP algorithm with time complexity sub-quadratic in n, we also need to quickly find the index j that minimizes the recurrence relations (3) and (4). Note that to get $F^k(i)$, we need to compute $\{G^p(\cdot), F^p(\cdot)\}$ for $p = 1, 2, \ldots, k$.

To motivate our approach, let us plot points $(G^k(j), R(j,i))$ in a 2-dimensional coordinate system for all j, $1 \leq j \leq i$, for a fixed vertex v_i. See Fig. 2. If we superimpose $G^k(j) + R(j,i) = c$ for a given value c in the same coordinate system, it is a $-45°$ line. If we increase c from 0, this line eventually touches one of the plotted points. The first point it touches gives the optimal value that minimizes $G^k(j) + R(j,i)$. In Fig. 2, this optimal is given by the point $(G^k(j_1), R(j_1,i))$. For convenience, let us refer to point $(G^k(j), R(j,i))$ as *point* (j,i).

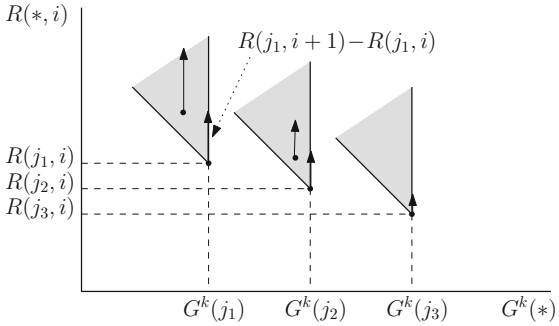

Fig. 2. $R(*,i)$ vs. $G^k(*)$. $j_1 < j_2 < j_3$.

We now explain that this representation provides us very useful information. To see it, for each point (j,i), define the *V-area* that lies above the $-45°$ line and to the left of the vertical line through it as shown as a shaded area in Fig. 2. We say that a point (j,i) situated in the V-area of another point (j_s,i) is *dominated* by (j_s,i), since the cost of point (j,i) is higher than the cost of (j_s,i). We sometimes say that v_j is dominated by v_{j_s}, when i is clear from the context. Thus the points at the bottoms of the V-areas are the only non-dominated points. For subpath $P[v_1,v_i]$ let $J(i) = \{j_1,\ldots,j_{g(i)}\}$, where $j_1 \leq j_2 \leq \ldots \leq j_{g(i)} \leq i$ and $\{(j_s,i) \mid s = 1,\ldots,g(i)\}$, are the set of all points at the bottoms of the V-areas. From the above discussion the following lemma follows directly.

Lemma 2. *$F^k(i) = G^k(j_1) + R(j_1,i)$ would hold if the path ended at vertex v_i.*

Function $G^k(i)$ can be computed in a similar manner. Let us now compare $J(i+1)$ for $P[v_1,v_{i+1}]$ with $J(i)$ for $P[v_1,v_i]$. Since $j_{g(i)} \leq i$, vertex v_{i+1} is farther from v_{j_s} than it is from v_{j_t}, if $s < t$. We thus have

$$R(j_s,i+1) - R(j_s,i) \geq R(j_t,i+1) - R(j_t,i) \quad \text{for } s < t. \tag{5}$$

The arrows in Fig. 2 indicate the increase $R(*,i+1) - R(*,i)$ in computing $J(i+1)$, compared with $J(i)$. Moreover, if (j,i) is dominated by (j_s,i), then point (j,i') will also be dominated by (j_s,i') for any $i' > i$. This implies that once it is determined that $(j,i) \notin J(i)$, then (j,i') will not belong to $J(i')$ for any $i' > i$. We will discuss how to update $J(i)$ to $J(i+1)$ in the next subsection.

3.2 Computing Switching Points

We compute $F^k(i)$ by Lemma 2, maintaining the set $J(i)$ of non-dominated candidate vertices. However, not all vertices in $J(i)$ that are non-dominated when computing $F^k(i)$ remain useful because some of these vertices may become dominated when computing $F^k(i')$ for $i' > i$. We shall identify those vertices, as they become dominated, and discard them.

Let us denote by $x(j_{s-1}, j_s)$, $1 < s \leq g(i)$, the *switching point*, namely the leftmost vertex $v_{i'}$ $(g(i) < i' \leq n)$, if any, for which j_s dominates j_{s-1}. If such an index does not exist, it means that j_s never dominates j_{s-1} and therefore we need not remember j_s. For convenience, let us introduce *a dummy vertex* j_0 so that we can write $x(j_0, j_1) = j_1$. Formally, we have

$$x(j_{s-1}, j_s) = \begin{cases} \min\{i' : [j_s < i' \leq n] \wedge [G^k(j_s) + R(j_s, i') \\ \qquad\qquad \leq G^k(j_{s-1}) + R(j_{s-1}, i')]\} & \text{if } s \geq 2, \\ j_s & \text{if } s = 1. \end{cases} \tag{6}$$

Computing and maintaining the sequence $x(j_0, j_1), \ldots, x(j_{g(i)-1}, j_{g(i)})$ allows us to determine a subset of non-dominated vertices v_{j_s}, which are potentially optimal vertices that may minimize function $F^k(\cdot)$ later. We therefore assume $x(j_0, j_1) < x(j_1, j_2) < \ldots < x(j_{g(i)-1}, j_{g(i)})$. To see this, assume for example that $x(j_2, j_3) < x(j_1, j_2)$ holds. Then j_2 will never be an optimal vertex, because for large enough i $(\geq x(j_1, j_2))$ which makes v_{j_2} dominate v_{j_1}, vertex v_{j_3} already dominates v_{j_2}, since $x(j_2, j_3) < x(j_1, j_2)$. This implies that j_2 can be removed from $J(i)$.

Updating set $J(i) = \{j_1, \ldots, j_{g(i)}\}$ to $J(i+1)$.

Change 1: If v_{j_1} becomes dominated by v_{j_2} (i.e., if $x(j_1, j_2) = i + 1$), then remove j_1 in constructing $J(i+1)$. (Note that $x(j_2, j_3) > i + 1$.)

Change 2: Starting from the last vertex in $J(i)$, find the rightmost vertex v_{j_s}, if any, that is not dominated by v_{i+1}. If none, let $s = 0$. Remove j_{s+1} to $j_{g(i)}$ in $J(i)$ to obtain $J(i+1)$. Put v_{i+1} in $J(i+1)$ as the last vertex $j_{g(i+1)}$.

It is easy to show that computing $J(i+1)$ from $J(i)$ takes amortized $O(t_X(n))$ time, where $t_X(n)$ is the time needed to compute $x(j, j')$ value for one arbitrary pair (j, j'), $j < j'$.

Based on the above discussion, we present Algorithm 1 below that shows a skeleton of our method for computing a minsum k-sink. In it, variable $J = \{j_1, j_2, \ldots, j_{g(i)}\}$ represents the ordered set of candidate vertices which is updated from iteration to iteration, and $X = \{x(j_0, j_1), x(j_1, j_2), \ldots, x(j_{g(i)-1}, j_{g(i)})\}$ represents the corresponding ordered set of switching points.

Lemma 3. *The minsum k-sink in dynamic flow path networks can be found in $O(kn \cdot t_X(n))$ plus preprocessing time.*

Algorithm 1. Minsum k-sink algorithm

1 **Input Data:** Number of evacuees w_i at each vertex i; Capacity c_i and length d_i for each edge e_i; An integer k representing the number of sinks to be located; $\mathcal{T}$ and $\mathcal{C}$ (defined in Sect. 4);
2 **Outputs:** A set $S^* \subseteq \{1, \ldots, n\}$ of k sinks to be located; Cost Z^* of solution S^*;
3 Base case: compute $G^1(i)$ for all $i \in \{1, \ldots, n\}$;
4 **for** $p \in \{1, \ldots, k\}$ **do**
5 $J \leftarrow 1; X \leftarrow 1;$ `//Initialize candidate sequence` J `and` $x(\emptyset, 1) = 1$
6 $F^p(1) \leftarrow G^p(1);$ `//` $L(1,1) = 0$
7 **for** $i \in \{2, \ldots, n\}$ **do**
8 **repeat** `//Update the candidate list` J `by considering` i
9 **if** J *is empty* **then**
10 $J \leftarrow i; X \leftarrow i;$ `//New vertex` v_i `is the dominating vertex`
11 *done* $\leftarrow$ `true`
12 **else**
13 $j \leftarrow$ last item in J; $x \leftarrow$ last item in X;
14 Compute $x(j, i)$; `//Switching point between` i `and` j
15 **if** $x(j, i)$ *does not exist* **then**
16 *done* $\leftarrow$ `true`; `//` i `is dominated`
17 **else if** $x(j, i) > x$ **then** `//` i `does not dominate` j
18 Append i at the end of J; append $x(j, i)$ at the end of X;
19 *done* $\leftarrow$ `true`
20 **else** `//` i `dominates` j
21 Remove j from the end of J; remove x from the end of X;
22 *done* $\leftarrow$ `false`
23 **end**
24 **end**
25 **until** *done*;
`// Check for Change 1 (in Sect. 3.2) and compute` $F^p(i)$
26 Let x^* be the rightmost vertex in X satisfying $x^* \leq i$ and let j^* be its corresponding vertex in J;
27 Let $F^p(i) \leftarrow G^p(j^*) + R(j^*, i)$
28 **end**
29 **if** $p < k$ **then**
30 Compute $G^{p+1}(i)$ in a similar way using $F^p(i)$ for all $1 \leq i \leq n$
31 **else**
32 **return** $Z^* = F^k(n)$; `//Sink set` S^* `can be obtained from` Z^* `in a standard way`
33 **end**
34 **end**

Proof. Algorithm 1 performs $O(kn)$ iterations in lines 4 and 7. The repeat loop at line 8 executes at most as many times as the size of list J. However, an element is added to list J at most once for each iteration i (lines 10 and 18), so J cannot receive more than n elements throughout the duration of the algorithm, and the repeat loop cannot have more than n iterations throughout the duration of the

algorithm. In each iteration of the repeat loop, $x(\cdot,\cdot)$ is computed a constant number of times, and the lemma follows. □

Now that we have the above lemma, the rest of this paper is devoted to making $t_x(n)$ as small as possible, culminating in Lemmas 7 and 8.

4 Data Structures for Computing $R(j,i)$ and $L(j,i)$

Costs $R(j,i)$ and $L(j,i)$ are used in (3), (4), and (6). It is needed wherever $x(j,i)$ is used in Algorithm 1, including Line 27. We only discuss how to compute $R(j,i)$, since $L(j,i)$ can be computed similarly. To compute $R(j,i)$, we need to know the section sequence of the vertices on $P[v_{j+1}, v_i]$ arriving at v_j. To find it efficiently, during preprocessing we construct a balanced binary tree, named *cluster tree* $\mathcal{T}$, whose leaves are the vertices of P, arranged from v_1 to v_n. We also construct a capacity tree $\mathcal{C}$, which is a standard binary search tree from which one can find capacity $c(v_j, v_h)$ in $O(\log n)$ time for any pair of vertices $v_j \preceq v_h$.

For each non-leaf node[2] u of $\mathcal{T}$, let $v_L(u)$ (resp. $v_R(u)$) denote the leftmost (resp. rightmost) vertex of P that belongs to subtree $\mathcal{T}(u)$. We say that u *spans* subpath $P[v_L(u), v_R(u)]$. For a node u of $\mathcal{T}$, let $\alpha_R^u(v_j)$ (resp. $\beta_R^u(v_j)$) denote the arrival (resp. departure) section sequence at (resp. from) v_j ($\preceq v_L(u)$), carrying the vertices spanned by u. At each node u of $\mathcal{T}$, we precompute and store $\alpha_R^u(v_L(u))$ and $\beta_R^u(v_L(u))$, as we describe below.

At a leaf node, which is a vertex v_i, it is easy to construct $\beta_R^{v_i}(v_i)$, which is just a section of height c_{i-1} and duration w_i/c_{i-1} that starts at time 0 (local time at v_i). We obviously have $\alpha_R^{v_i}(v_i) = \emptyset$. For an internal node u of $\mathcal{T}$ with two child nodes, u_a and u_b, assuming that we have computed $\alpha_R^{u_a}(v_L(u_a))$, $\beta_R^{u_a}(v_L(u_a))$, $\alpha_R^{u_b}(v_L(u_b))$, and $\beta_R^{u_b}(v_L(u_b))$, we want to compute $\alpha_R^u(v_L(u))$ and $\beta_R^u(v_L(u))$ from them. Let $v_{j+1} = v_L(u_a)$. Then $\beta_R^{u_b}(v_L(u_b))$ would become $\alpha_R^{u_b}(v_{j+1})$ with a delay of $d(v_{j+1}, v_L(u_b))\tau$ according to the local time at v_{j+1}, provided it encountered no congestion on its way. If the height of an arriving section in $\alpha_R^{u_b}(v_{j+1})$ is larger than c_j, the evacuees carried by that section cannot depart from v_{j+1} at the arrival rate. See Fig. 3(a), where $S_1, S_2, \ldots$, are the sections of $\alpha_R^{u_b}(v_{j+1})$, arriving at v_{j+1}. The durations of some sections get stretched in this case, by the *ceiling* operation [12]. The following two situations can arise to these sections, when they are converted into the departing sections of $\beta_R^u(v_{j+1})$. (When S_1 arrives, there may be w (> 0) leftover evacuees at the vertex. We will consider such a scenario shortly.)

(a) A stretched section by a ceiling operation ends in a gap. (Fig. 3(b) shows that the stretched S_1 fills the next gap entirely, merges with S_2, and the following gap is partially filled. The amount equal to the light-gray parts consisting of later arrivals moves to the dark-gray parts, to fill gaps.)
(b) A section may shrink due to the expanded section preceding it, with its front pushed to a later time. (In Fig. 3(c), the stretched S_3 "swallows" a part of

[2] We use the term "node" for $\mathcal{T}$ to distinguish them from the vertices of P.

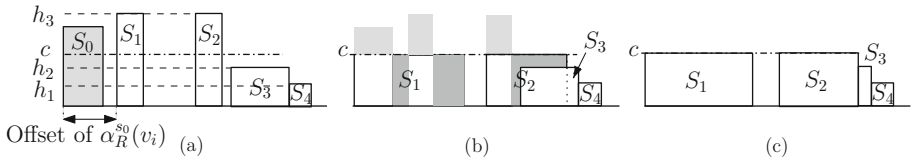

Fig. 3. (a) $\alpha_R^{u_b}(v_{j+1})$; (b) Amount equal to the light-gray parts fill the dark-gray parts; (c) Result.

S_4 and S_4 shrinks. The next section (such as S_5 in this example), if any, undergoes no change, since its height is less than c_j.)

From observations (a) and (b) above, we can easily infer the following lemma.

Lemma 4. *The heights of the sections in $\alpha_R^u(v_{j+1})$ and $\beta_R^u(v_{j+1})$ are non-increasing with time.*

Redefine $S_1, S_2, \ldots,$ to be the sections of $\alpha_R^u(v_L(u))$. To find which sections merge with other sections when a smaller capacity is encountered, we place the weight-time ratios

$$\{\lambda(S_h)/\delta_h \mid S_h \text{ is a section of } \alpha_R^u(v_L(u))\}$$

in a max-heap $\mathcal{H}_u$, where $\lambda(S_h)$ is the sum of the weights of the vertices carried by S_h, and δ_h is the time difference between the fronts of S_h and S_{h+1}. Thus in converting $\alpha_R^u(v_{j+1})$ into $\beta_R^u(v_{j+1})$, we pop out of $\mathcal{H}_u$ those ratios that are larger than c_j, and for each such ratio, we merge the corresponding pair of sections. In Fig. 3, for example, $\lambda(S_3)/\delta_3 > \lambda(S_1)/\delta_1 > c_j$, so that S_3 merges with S_4, and S_1 merges with S_2. For the resulting new sections, we compute the weight-time ratios, and if they are larger than c_j, we repeat the merging process.

We call two nodes u_a and u_b of $\mathcal{T}$ *adjacent* if $v_R(u_a)$ and $v_L(u_b)$ are adjacent vertices on P. It is easy to observe

Proposition 1. *Let u_a and u_b be two adjacent nodes of $\mathcal{T}$. The evacuees still left at vertex $v_L(u_a)$, if any, when the first evacuee in $\alpha_R^{u_b}(v_{j+1})$ (shifted $\beta_R^{u_b}(v_L(u_b))$) arrives there belong to the last cluster in $\beta_R^{u_a}(v_L(u_a))$.*

We say those leftover evacuees form a *backlog*. If the height of an incoming section is less than c_j, then we use the underutilized capacity to accommodate as many of the delayed backlog evacuees as possible, together with the evacuees carried by the section. See Fig. 4, where the area of the dark-gray part equals the backlog. In this example, a section of height c_j becomes a new section in the departure section sequence out of v_{j+1}, and the light-gray part of S_3 is also a new section. Sections S_4 and S_5 maintain their shapes as they go through v_{j+1}.

At vertex $v_{j+1} = v_L(u_a)$, the first cluster in $\alpha_R^{u_b}(v_{j+1})$ would start at δ time units before all the evacuees at v_{j+1} from $P[v_L(u_a), v_R(u_a)]$ would have left, if there was no congestion on its way. Thus there would be a backlog of $w = c_j\delta$

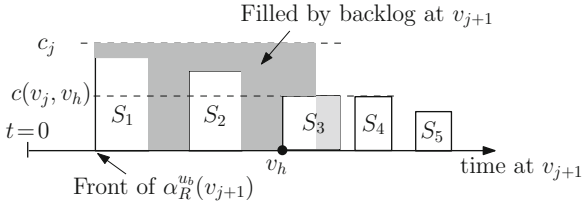

Fig. 4. $\alpha_R^{u_b}(v_{j+1})$ undergoes a change to become a part of the departure section sequence out of v_{j+1}.

evacuees still waiting at v_{j+1}. We use w arriving evacuees to fill the "space" (gaps and underutilized capacities) among initial sections in $\alpha_R^{u_b}(v_{j+1})$, as we stated before. Let $S_1, S_2, \ldots,$ be the sections of $\alpha_R^{u_b}(v_{j+1})$, which is already ceiled by c_j, and let S_h start at time t_h. Then the total amount of "space" between the first and the l^{th} sections is filled by w if

$$w \geq (t_l - t_1)c_j - \sum_{h=1}^{l-1} \lambda(S_h). \tag{7}$$

We test $l = 1, 2, \ldots$ sequentially to find up to which gap gets merged due to the backlog. The last gap (which may be after S_l and semi-infinite) is generally only partially filled.

Lemma 5. *We can construct $\mathcal{T}$ (with $\alpha_R^u(v_L(u))$ and $\beta_R^u(v_L(u))$ at every node u) in $O(n \log^2 n)$ time.*

Proof. For a node u of $\mathcal{T}$ with two child nodes u_a and u_b, we discussed above how to compute $\alpha_R^u(v_L(u))$ and $\beta_R^u(v_L(u))$, given $\alpha_R^{u_a}(v_L(u_a))$, $\beta_R^{u_a}(v_L(u_a))$, $\alpha_R^{u_b}(v_L(u_b))$, and $\beta_R^{u_b}(v_L(u_b))$. The ceiling operation of $\beta_R^{u_b}(v_L(u_b))$ by c_j, using max-heap $\mathcal{H}_u$, takes $O(|\mathcal{T}(u)| \log |\mathcal{T}(u)|)$ time, since each insertion into $\mathcal{H}_u$ takes $O(\log |\mathcal{T}(u)|)$ time, where $|\mathcal{T}(u)|$ denotes the number vertices spanned by u. The sequential tests, using (7), to find the extent of gap filling takes $O(|\mathcal{T}(u)|)$ time. Thus the total time for all nodes is $O(n \log^2 n)$. □

From now on we assume that $\mathcal{T}$ is constructed during preprocessing and available. We will make use of it in proving Lemma 6.

5 Putting Pieces Together

5.1 Computing $R(j, i)$

To run our DP, we need to compute cost $R(j, i) = \sum_h (E_h + I_h)$, where E_h (resp. I_h) is the extra cost (resp. intra cost), defined by (2), of arrival section S_h at v_{j+1}, which carries vertices of a subpath of $P[v_{j+1}, v_i]$.

Lemma 6. *If $\mathcal{T}$ is given, then for an arbitrary pair (j,i), $j<i$, we can compute $R(j,i)$ in $O((i-j)\log n)$ time.*

Proof. Let $\mathcal{P}[v_{j+1},v_i]$ denote the set of maximal subpaths of $P[v_{j+1},v_i]$ spanned by $t=O(\log n)$ nodes, $u_1,u_2,\ldots,u_t$, of $\mathcal{T}$, in this order from left to right. To compute $R(j,i)$, we combine the arrival and/or departure section sequences stored at $u_1,u_2,\ldots,u_t$ into a single arrival sequence at v_j. We discussed in Sect. 4 in detail how to combine two such sequences. Starting with $\sigma=\alpha_R^{u_1}(v_L(u_1))$, we update σ by merging it with shifted $\beta_R^{u_2}(v_L(u_2)),\beta_R^{u_3}(v_L(u_3)),\ldots$, until all of them are merged into one arrival section sequence at v_j. The shift amount for $\beta_R^{u_s}(v_L(u_s))$ is $d(v_L(u_1),v_L(u_s))\tau$. When we merge σ with $\beta_R^{u_s}(v_L(u_s))$, the capacity $c(v_j,v_L(u_s))$ must be used to ceil the shifted $\beta_R^{u_s}(v_L(u_s))$. Finding this capacity from the capacity tree $\mathcal{C}$ takes $O(\log n)$ time. The most time consuming part is testing (7) for successive l, every time two section sequences are merged. Since we must perform $O(i-j)$ such tests, the total time for all the merges is $O((i-j)\log n)$. Once the arrival section sequence at v_j is known, we can compute the intra and extra cost based on (2). □

Lemma 7. *Assuming that $\mathcal{T}$ is available, we have $t_X(n)=O(n\log^2 n)$.*

Proof. Evaluating $R(\cdot,\cdot)$ takes $O((i-j)\log n)$ time by Lemma 6, and the total time for finding switching point $x(j,j')$ is $O(n\log^2 n)$, since we need to perform binary search. □

Lemma 8. *If the edge capacities are uniform, we have $t_X(n)=O(\log^3 n)$.*

Proof. We precompute at each node u of $\mathcal{T}$ the sum of squared weights for the sections carrying the vertices spanned by u. In processing the backlog to fill gaps between the sections, the contributions from the swallowed up sections are subtracted from, and the squared weight of the new combined section is added to the sum of squared weights. Thus updating the sum of squared weights takes constant time per merging two adjacent subtrees spanning subpaths in $\mathcal{P}[v_{j+1},v_i]$. In the general capacity case, we tested (7) for successive l sequentially. But in the uniform capacity case, we can maintain the prefix sum of the gaps between successive sections. Then we can do binary search among them with backlog w to find up to which gaps are filled by w. This takes $O(\log n)$ time per merger of section sequences stored at two adjacent nodes of $\mathcal{T}$, and $O(\log^2 n)$ time for all such mergers. Thus we can find $R(j,i)$ in $O(\log^2 n)$ time, hence $t_X(n)=O(\log^3 n)$. □

5.2 Main Theorem

The correctness of our DP method can be proved similarly to [7] and the discussions above. Time complexities were analyzed in Lemmas 3, 5, 7, and 8.

Theorem 1. *(a) The minsum k-sink problem in dynamic flow path networks can be solved in $O(kn^2\log^2 n)$ time.*
(b) If the edge capacities are uniform, then it can be solved in $O(kn\log^3 n)$ time.

6 Conclusion and Discussion

We proposed an $O(kn^2 \log^2 n)$ time algorithm, based on DP that finds a minsum k-sink in dynamic flow path networks with general edge capacities, which is the first polynomial time algorithm for this problem. When the edge capacities are uniform, we also presented an $O(kn \log^3 n)$ time algorithm. There is a factor of n difference between the above two cases. The main reason is that in the general capacity case, we cannot update the intra cost in less than linear time, when merging two section sequences. We are currently working to find a way around it. A challenging problem is to efficiently solve the minsum k-sink problem in dynamic flow networks that are more general than path networks.

References

1. Arumugam, G.P., Augustine, J., Golin, M., Srikanthan, P.: A polynomial time algorithm for minimax-regret evacuation on a dynamic path. arXiv:1404,5448v1 [cs.DS], 22 April 2014, 165 (2014)
2. Bhattacharya, B., Golin, M.J., Higashikawa, Y., Kameda, T., Katoh, N.: Improved algorithms for computing k-sink on dynamic flow path networks. In: Ellen, F., Kolokolova, A., Sack, J.R. (eds.) Algorithms and Data Structures. LNCS, vol. 10389, pp. 133–144. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-62127-2_12
3. Bhattacharya, B., Kameda, T.: Improved algorithms for computing minmax regret sinks on path and tree networks. Theoret. Comput. Sci. **607**, 411–425 (2015)
4. Chen, D., Golin, M.: Sink evacuation on trees with dynamic confluent flows. In: Hong, S.-H. (ed.) 27th International Symposium on Algorithms and Computation (ISAAC), Leibniz International Proceedings in Informatics, pp. 25:1–25:13 (2016)
5. Cheng, S.-W., Higashikawa, Y., Katoh, N., Ni, G., Su, B., Xu, Y.: Minimax regret 1-sink location problems in dynamic path networks. In: Chan, T.-H.H., Lau, L.C., Trevisan, L. (eds.) Theory and Applications of Models of Computation. LNCS, vol. 7876, pp. 121–132. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-38236-9_12
6. Hamacher, H., Tjandra, S.: Mathematical modelling of evacuation problems: a state of the art. In: Pedestrian and Evacuation Dynamics, pp. 227–266. Springer (2002)
7. Hassin, R., Tamir, A.: Improved complexity bounds for location problems on the real line. Oper. Res. Lett. **10**(7), 395–402 (1991)
8. Higashikawa, Y., Augustine, J., Cheng, S.W., Golin, M.J., Katoh, N., Ni, G., Su, B., Xu, Y.: Minimax regret 1-sink location problem in dynamic path networks. Theoret. Comput. Sci. **588**(11), 24–36 (2015)
9. Higashikawa, Y., Golin, M.J., Katoh, N.: Minimax regret sink location problem in dynamic tree networks with uniform capacity. J. Graph Algorithms Appl. **18**(4), 539–555 (2014)
10. Higashikawa, Y., Golin, M.J., Katoh, N.: Multiple sink location problems in dynamic path networks. Theoret. Comput. Sci. **607**(1), 2–15 (2015)
11. Kariv, O., Hakimi, S.: An algorithmic approach to network location problems, Part II: the p-median. SIAM J. Appl. Math. **37**, 539–560 (1979)
12. Mamada, S., Uno, T., Makino, K., Fujishige, S.: An $O(n \log^2 n)$ algorithm for a sink location problem in dynamic tree networks. Discrete Appl. Math. **154**, 2387–2401 (2006)