A Polynomial Time Algorithm for Minimax-Regret Evacuation on a Dynamic Path

Guru Prakash Arumugam\textsuperscript{1} John Augustine\textsuperscript{1} Mordecai Golin\textsuperscript{2} Prashanth Srikanthan\textsuperscript{1}

\textsuperscript{1} Indian Institute of Technology Madras
\textsuperscript{2} Hong Kong University of Science and Technology

Abstract. A dynamic path network is an undirected path with evacuees situated at each vertex. To evacuate the path, evacuees travel towards a designated sink (doorway) to exit. Each edge has a capacity, the number of evacuees that can enter the edge in unit time. Congestion occurs if an evacuee has to wait at a vertex for other evacuees to leave first. The basic problem is to place \( k \) sinks on the line, with an associated evacuation strategy, so as to minimize the total time needed to evacuate everyone.

The minmax-regret version introduces uncertainty into the input, with the number of evacuees at vertices only being specified to within a range. The problem is to find a universal solution whose regret (difference from optimal for a given input) is minimized over all legal inputs.

The previously best known algorithms for the minmax-regret version problem ran in time exponential in \( k \). In this paper, we derive new properties of solutions that yield the first polynomial time algorithms for solving the problem.

1 Introduction

Dynamic flow networks model movement of items on a graph. The process starts with each vertex \( v \) assigned some initial set of supplies \( w_v \). Supplies flow across edges from one vertex to another. Each edge \( e \) has a given capacity \( c_e \) which limits the rate of the flow of supplies into the edge in each time unit. If all edges have the same capacity \( c_e = c \) the network has uniform capacity. Each \( e \) also has a time required to travel the edge. Note that as supplies move around the graph, congestion can occur, as supplies back up at a vertex.

Dynamic flow networks were introduced in [3] and have since been extensively analyzed. One well studied problem on such networks is the transshipment problem, e.g., [6], in which the graph has several sources and sinks, with the original supplies being the sources and each sink having a specified demand. The problem is then to find the minimum time required to satisfy all of the demands.

Dynamic Flow problems also model evacuation problems. In these, the vertex supplies are people in a building(s) and the problem is to find a routing strategy (evacuation plan) that evacuates all of them to specified sinks (exits) in minimum time. Note that in these problems the evacuation plan is vertex based.
That is, each vertex has one associated evacuation edge; all people starting
or arriving at that vertex must evacuate through that edge. After traversing
the edge they arrive at another vertex and traverse its evacuation edge. This
continues until a sink is reached and the people exit. The basic optimization
problem is to determine a plan that minimizes the total time needed to evacuate
all the people.

In some versions of the problem the sinks are known in advance. In others,
such as the ones we will address in this paper, the placement of the sink(s) is
part of the evacuation plan with only \( k \), the number of allowed sinks, being
specified as part of the problem. To the best of our knowledge there is no known
polynomial time algorithm for solving this problem on a general graph. [10]
gives an \( O(n \log^2 n) \) algorithm for solving the 1-sink problem on a dynamic tree
network with general capacities. [5] improves this down to to \( O(n \log n) \) when the
capacities are uniform. [4] shows how to solve the \( k \)-sink problem on a uniform
capacity dynamic path network in \( O(kn \log n) \) time.

In practice, the exact input, e.g., number of people \( w_v \) at each vertex \( v \),
is unknown at the time the plan is drawn up. All that may be known is that
\( w_v \in [w_v^-, w_v^+] \) for some specified range.

One model for attacking this type of uncertainty is to define the regret of a
plan on a particular fixed input as the discrepancy between the evacuation time
for that plan on that input and the minimum time needed to evacuate for that
input. The maximum regret of the plan is then taken over all possible inputs.
The minmax-regret plan is the one that minimizes the maximum regret.

Minmax regret optimization has been extensively studied for the \( k \)-median
([1] is a recent case) and many other optimization problems ([8] provides an
introduction to the literature). The 1-sink minmax-regret evacuation problem
on a uniform capacity path was originally solved in \( O(n \log^2 n) \) time by [2]. This
was reduced down to \( O(n \log n) \) by [12]. [4] provides an \( O(n \log^2 n) \) algorithm
for the 1-sink minmax-regret problem on a uniform capacity tree.

Returning to the minmax-regret problem uniform capacity path case, the
only algorithms known for \( k > 1 \) were [9] which for \( k = 2 \) gave an \( O(n^3 \log n) \)
algorithm and [11] which gave an \( O(n^{1+k} \log n)^{1+\log k} \) algorithm for general \( k \).

In this paper we derive many new properties of the the minmax-regret uni-
form capacity path problem. These lead to two new algorithms, both of which
improve on the previous ones. The first, which is better for small \( k \), runs in
\( O(kn^2(\log n)^k) \). The second, better for larger \( k \), runs in \( O(kn^3 \log n) \), which is
the first polynomial time algorithm for this problem.

The paper is structured as follows. Section 2 introduces the setting and defi-
nitions. Section 3 discusses how to solve the optimal non-regret problem (whose
solution will be needed later). Section 4 derives many of the minmax regret
properties needed. Section 5 gives our first algorithm, good for small \( k \). Section
6 derives more properties. Section 7 gives our second, polynomial time, algo-

2
2 Preliminary Definitions

2.1 Model Definition

Consider a path \( P = (V, E) \) with \((n + 1)\) vertices(buildings) \( V = \{x_0, x_1, ..., x_n\} \) and \( n \) edges(roads) \( E = \{e_1, e_2, ..., e_n\} \) with each edge \( e_i = (x_{i-1}, x_i) \). \( x_i \) also denotes the line-coordinate of the \( i^{th} \) building and \( x_0 < x_1 < ... < x_n \).

The distance between two vertices \( x_i \) and \( x_j \) is given by \( |x_i - x_j| \). Each vertex \( x_i \in V \) is associated with an open interval of weights \( [w_i^-, w_i^+] \), denoting the range within which the number of evacuees in building \( x_i \) can lie (\( 0 < w_i^- \leq w_i^+ \)).

Each edge has a capacity \( c \). We define \( \tau \) as the time taken to travel a unit distance on the edge.

Let \( S \) denote the Cartesian product of all weight intervals for \( 0 \leq i \leq n \):

\[
S = \prod_{0 \leq i \leq n} [w_i^-, w_i^+].
\]

A scenario \( s \in S \) is an assignment of weights (number of evacuees) to all vertices. The weight of a vertex \( x_i \) under scenario \( s \) is denoted by \( w_i(s) \).

In the paper, we refer to a function \( f(x) \) as “unimodal with a unique minimum value” if there exists an \( m \) such the function \( f(x) \) is monotonically decreasing for \( x \leq m \) and monotonically increasing for \( x \geq m \). The minimum value of the function is attained for \( x = m \).

2.2 Evacuation Time

Consider a path \( P = (V, E) \), with uniform edge capacity \( c \), and time to travel unit distance \( \tau \). We are given a scenario \( s \in S \), i.e., vertex \( x_i \) has weight \( w_i(s) \).

Left and Right Evacuation

Consider a single sink \( x \in P \).

If there were no other people in the way, the evacuees at \( v_i \) could complete evacuating to \( x \) in time \( (|x_i - x| \tau + \lceil w_i(s)/c \rceil - 1) \).

In reality, people to the left and right of the sink \( x \) need to evacuate to \( x \) (as shown in Fig. 1a) and the actual time required must take congestion into account.

We define \( \Theta_L(P, x, s) \) (resp. \( \Theta_R(P, x, s) \)) to be the time taken by people to the left(resp. right) of sink \( x \) to evacuate to \( x \) under scenario \( s \). The left(resp. right) evacuation time can be expressed as the maximum of an evacuation function of the nodes to the left(resp. right) of the sink. The exact expression, taken from [7] are as follows:

\[
\Theta_L(P, x, s) = \max_{x_i < x} \left\{ (x - x_i)\tau + \left\lceil \frac{\sum_{0 \leq j \leq i} w_j(s)}{c} \right\rceil - 1 \right\} \tag{1}
\]

\[
\Theta_R(P, x, s) = \max_{x_i > x} \left\{ (x_i - x)\tau + \left\lceil \frac{\sum_{i \leq j \leq n} w_j(s)}{c} \right\rceil - 1 \right\} \tag{2}
\]
where the maximum is taken over the evacuation function of every node.

For exposition, we simplify the equations (as done in [2,11]) by assuming $c = 1$ and omitting the constant (i.e., -1). Setting $c = 1$ will help us simplify some of the proofs later in the paper (but note that the case of $c > 1$ can be treated essentially in the same manner). Thus the left and right evacuation times, $\Theta_L(P,x,s)$ and $\Theta_R(P,x,s)$, are redefined as follows:

$$\Theta_L(P,x,s) = \max_{x_i < x} \left\{ (x - x_i)\tau + \sum_{0 \leq j \leq i} w_j(s) \right\}$$

$$\Theta_R(P,x,s) = \max_{x_i > x} \left\{ (x_i - x)\tau + \sum_{i \leq j \leq n} w_j(s) \right\}$$

**1-Sink Evacuation** The evacuation time to sink $x$ is the maximum of the left and right evacuation times, $\Theta^1(P,x,s) = \max \{ \Theta_L(P,x,s), \Theta_R(P,x,s) \}$. The superscript “1” denotes that this is a 1-Sink problem.

**k-Sink Evacuation** Naturally extending the 1-Sink evacuation, suppose we are given a $k$-partition of the path $P$ into $k$ subpaths (or parts) $\tilde{P} = \{P_1, P_2, ..., P_k\}$ and a set of sinks $\tilde{Y} = \{y_1, y_2, ..., y_k\}$ such that each $y_i \in P_i$. The evacuees in part $P_i$ are restricted to evacuate only to sink $y_i$ (See Fig. 1b). Every node should completely belong to one part, i.e., no $k$-partition is allowed to split a node. This restricts the people in a building to evacuate to one sink in their part without confusion. The $k$-sink evacuation time will essentially be the maximum of the evacuation time among the individual parts.

Let $\Theta^k(P,\{\tilde{P},\tilde{Y}\},s)$ denote the $k$-sink evacuation time on path $P$ with $k$-partition and sinks $\{\tilde{P},\tilde{Y}\}$ under scenario $s$. Since the evacuation in each subpath $P_i$ are independent of each other, $\Theta^k(P,\{\tilde{P},\tilde{Y}\},s)$ can be expressed as:

$$\Theta^k(P,\{\tilde{P},\tilde{Y}\},s) = \max_{1 \leq i \leq k} \Theta^1(P_i,y_i,s)$$

**Definition 1.** Let $d$ be the smallest (and therefore unique) index among the indices of the part(s) which maximizes the term $\Theta^1(P_i,y_i,s)$. The evacuation
time of subpath $P_d$ determines the $k$-Sink evacuation time under scenario $s$. We refer to $P_d$ as the “dominant part” under scenario $s$.

2.3 Optimal (Non-Regret) $k$-Sink Location Problem

The Optimal $k$-Sink Location Problem can be defined as follows. The input is a path $P$ with edge capacities $c$, time to travel unit distance $\tau$, and a scenario $s \in S$.

Let $\Theta_{opt}(P, s)$ be the optimal (minimum) $k$-Sink evacuation time in path $P$ under scenario $s \in S$. The algorithm needs to provide the $k$-partition and sinks $\{\hat{P}, \hat{Y}\}$ as defined in Sect. 2.2 which achieves this minimum evacuation time.

2.4 Regret

For a choice of $k$-partition and sinks $\{\hat{P}, \hat{Y}\}$ and a scenario $s \in S$, the regret is defined as the difference between the $k$-Sink evacuation time for this choice of $\{\hat{P}, \hat{Y}\}$ and the optimal $k$-Sink evacuation time. The regret can be expressed as:

$$R(\{\hat{P}, \hat{Y}\}, s) = \Theta^k(P, \{\hat{P}, \hat{Y}\}, s) - \Theta_{opt}^k(P, s)$$

(6)

The maximum regret (called max-regret) achieved (over all scenarios) for a choice of $\{\hat{P}, \hat{Y}\}$ is:

$$R_{max}(\{\hat{P}, \hat{Y}\}) = \max_{s \in S} \left\{ R(\{\hat{P}, \hat{Y}\}, s) \right\}$$

(7)

If $R_{max}(\{\hat{P}, \hat{Y}\}) = R(\{\hat{P}, \hat{Y}\}, s^*)$ for some scenario $s^* \in S$, then $s^*$ is called a worst-case scenario for $\{\hat{P}, \hat{Y}\}$. The minimax-regret for the problem is the minimum max-regret over all possible $k$-partitions and sinks $\{\hat{P}, \hat{Y}\}$.

2.5 Minimax-Regret $k$-Sink Location Problem

The input for the Minimax-Regret $k$-Sink Location Problem is a dynamic path network with path $P$, vertex weight intervals $[w_i^-, w_i^+]$, edge capacity $c$, and time to travel unit distance $\tau$. The problem can be understood as a 2-person game between the algorithm $A$ and the adversary $B$ as follows:

1. The algorithm $A$ provides a $k$-partition and sinks $\{\hat{P}, \hat{Y}\}$ as defined in Sect. 2.2.
2. The adversary $B$ now chooses a regret-maximizing worst-case scenario $s^* \in S$ with a max-regret of $R_{max}(\{\hat{P}, \hat{Y}\})$.
3. The objective of $A$ is to find the $k$-partition and sinks $\{\hat{P}, \hat{Y}\}$ that minimizes the max-regret.
2.6 Sink on Vertex Assumption

All our definitions until now are generic in the fact that a sink can lie either at a vertex or on an edge (i.e., anywhere on the path). From this point onwards, we are going to assume that a sink always lies at a vertex (and minmax-regret is defined under this assumption). This implies that the end-points of a part should also lie at vertices (since there can be no sink on an edge). For example, if $P_i$ and $P_{i+1}$ are consecutive parts and the right end-point of $P_i$ is some vertex $x_{r_i}$, then the left end-point of $P_{i+1}$ will be $x_{r_i+1}$ and the edge between $x_{r_i}$ and $x_{r_i+1}$ does not take part in the evacuation.

Note that although we are making the sink on vertex assumption, the properties stated later in the paper all hold even without the assumption. With extra complexity, our algorithms could be extended to work without this assumption.

3 A Solution to the Optimal (Non-Regret) $k$-Sink Location Problem

In this section, we wish to show an $O(kn \log n)$ time solution to the optimal $k$-sink location problem. In our problem, the flow (of people) to the sink is discrete, since the objects of evacuation (people) are discrete entities. The left and right evacuation times to a sink for this discrete flow was defined in Eqs. 1 and 2.

We note that [5] has given an $O(kn \log n)$ algorithm for the optimal $k$-sink location problem, but it was for the continuous version of the problem (the object of evacuation is continuous like a fluid). They derive the following formula for the left evacuation time to a sink $x$ as:

$$\Theta_L(P, x, s) = \max_{x_i < x} \left\{ (x - x_i)\tau + \sum_{0 \leq j \leq i} w_j(s) \right\}$$

and the right evacuation time is defined similarly.

As we can see from Eq. 8 the expressions for the continuous version does not involve a ceiling function. The method used in [5] is not extendible for the discrete case due to complications arising from the ceiling function in the expression.

We observe some key properties of the optimal solution in the discrete case and using a novel data structure which we call the Bi-Heap, a dynamic programming based procedure has been proposed and explained in detail in Appendix A. The running time of the algorithm is $O(kn \log n)$.

4 Properties of the Minimax-Regret $k$-Sink Location Problem

We now elaborate on a few properties of the Minimax-Regret $k$-Sink Location Problem which will help us come up with efficient algorithms.
4.1 Structure of the worst-case scenario

We note that the adversary $B$ needs to choose a worst-case scenario but is free to choose any worst-case scenario since all of them give the same max-regret. In this section, Lemmas 1 and 2 are going to show that there always exists a worst-case scenario with a particular structure and we will assume without loss of generality that $B$’s chosen worst-case scenario has this structure.

Now, assume that the algorithm $A$ has given $k$-partition and sinks $\hat{P}, \hat{Y}$.  

**Lemma 1.** Let $s_B^* \in S$ be a worst-case scenario with its dominant part $P_d \in \hat{P}$. Then if $s_B^*$ is modified so that $w_i(s_B^*) = w_i^-$ if $x_i \notin P_d$ (See Fig. 2a), $s_B^*$ remains a worst case scenario. (Proof in Appendix D.1.)

**Definition 2.** Given a scenario, a “sub-scenario” for a subpath is defined as the scenario within that subpath.

**Definition 3.** Consider a subpath $P'$ of the path $P$ with leftmost(resp. right-most) vertex $x_l$ (resp. $x_r$). Under some scenario $s \in S$ a sub-scenario for $P'$ is called left-dominant(resp. right-dominant) if for some $i$ with $l \leq i \leq r$, $w_j(s) = w_j^+$ (resp. $w_j(s) = w_j^-$) for $l \leq j < i$ and $w_j(s) = w_j^-$ (resp. $w_j(s) = w_j^+$) for $i \leq j \leq r$.

Given a $k$-partition $\hat{P} = \{P_1, P_2, ..., P_k\}$, let $S_i^L$ (resp. $S_i^R$) denote the set of all left-dominant (resp. right-dominant) sub-scenarios in the part $P_i$.

**Lemma 2.** There exists a worst-case scenario $s_B^*$ with its dominant part $P_d \in \hat{P}$ such that the sub-scenario within $P_d$ lies in the set $S_d^L \cup S_d^R$ (See Fig. 2b.) (Proof in Appendix D.2.)

**Theorem 1.** For any choice of $k$-partition and sinks, $\{\hat{P}, \hat{Y}\}$, made by algorithm A, there exists a worst-case scenario $s_B^* \in S$ with its dominant part $P_d \in \hat{P}$ of the following form:

1. $w_i(s_B^*) = w_i^-$ if $x_i \notin P_d$ and,
2. Sub-scenario in $P_d$ is in the set $S_d^L \cup S_d^R$.

**Proof.** Trivially follows from Lemmas 1 and 2.
Let us denote by $S^* \subseteq S$, the set of all possible worst-case scenarios of the form defined in Theorem 1. Without loss of generality, we assume that the adversary $B$ only chooses a worst-case scenario $s^*_B \in S^*$.

Property 1. Given $k$ parts and sinks $\{\hat{P}, \hat{Y}\}$ (or only the parts $\hat{P}$) by algorithm $A$, there are $O(n)$ candidate worst-case scenarios from which $B$ chooses $s^*_B \in S^*$. (Proof in Appendix D.3)

Property 2. Overall, there are only $O(n^2)$ possible candidate worst-case scenarios for $s^*_B$ irrespective of the $k$-partition/sinks given by the algorithm, i.e., $|S^*| = O(n^2)$. (Proof in Appendix D.4)

4.2 Characterization of Minimax-Regret

Suppose algorithm $A$ chooses $k$-partition and sinks $\{\hat{P}, \hat{Y}\}$, we will assume $x_{l_i}$ (resp. $x_{r_i}$) to be the left end (resp. right end) of the $i^{th}$ part $P_i \in \hat{P}$. For any scenario $s \in S$, the regret can be written as:

$$R(\{\hat{P}, \hat{Y}\}, s) = \Theta^k(P, \{\hat{P}, \hat{Y}\}, s) - \Theta^k_{\text{opt}}(P, s) \quad (\text{from Eq. 6})$$

$$= \max_{1 \leq i \leq k} \left\{ \Theta^1(P_i, y_i, s) - \Theta^k_{\text{opt}}(P, s) \right\} \quad (\text{from Eq. 5})$$

$$= \max_{1 \leq i \leq k} \left\{ \Theta^1(P_i, y_i, s) - \Theta^k_{\text{opt}}(P, s) \right\}$$

$$= \max_{1 \leq i \leq k} \left\{ R_{l_i r_i}(s, y_i) \right\}. \quad (9)$$

The term $R_{l_i r_i}(s, y_i)$ refers to the regret under scenario $s \in S$ when the part $P_i$ is assumed to be the dominant part and $y_i$ is the sink in $P_i$. We refer to $P_i$ as the Assumed Dominant Part (ADP). In reality there is only one unique dominant part, say $P_d$, under scenario $s$. It is easy to see that the maximum is achieved for this dominant part $P_d$, i.e., when $i = d$.

The max-regret for $A$’s choice of $\{\hat{P}, \hat{Y}\}$ can be written as:

$$R_{\text{max}}(\{\hat{P}, \hat{Y}\}) = \max_{s \in S^*} \left\{ R(\{\hat{P}, \hat{Y}\}, s) \right\} \quad (\text{from Eq. 7})$$

$$= \max_{s \in S^*} \max_{1 \leq i \leq k} \left\{ R_{l_i r_i}(s, y_i) \right\} \quad (\text{from Eq. 9})$$

$$= \max_{1 \leq i \leq k} \left\{ \max_{s \in S^*} \left\{ R_{l_i r_i}(s, y_i) \right\} \right\}$$

$$= \max_{1 \leq i \leq k} \left\{ R_{l_i r_i}(y_i) \right\}. \quad (10)$$

The term $R_{l_i r_i}(y_i)$ denotes the maximum regret over all candidate worst-case scenarios if $P_i$ were the ADP with sink $y_i$. This characterization of the max-regret, $R_{\text{max}}(\{\hat{P}, \hat{Y}\})$, shows the independence of the parts for max-regret calculations.
The minimax-regret can be written as:

\[
\min_{\{P, Y\}} R_{\max}(\{\hat{P}, \hat{Y}\}) = \min_{\{P, Y\}} \max_{1 \leq i \leq k} \{ R_{l_i r_i}(y_i) \} \quad \text{(from Eq. 10)}
\]

\[
= \min_P \max_{1 \leq i \leq k} \left\{ \min_Y R_{l_i r_i}(y_i) \right\} \quad \text{(11)}
\]

\[
= \min_P \max_{1 \leq i \leq k} \{ R_{l_i r_i} \} . \quad \text{(12)}
\]

The term \( R_{l_i r_i} \) denotes the minimax-regret over all candidate worst-case scenarios if \( P_i \) were the ADP. The minimax-regret for ADP \( P_i \) can be thought of as iterating over all possible sinks \( y_i \in P_i \) and choosing the sink which minimizes the max-regret \( R_{l_i r_i}(y_i) \).

We now state two important lemmas which will be used in further proofs.

**Lemma 3.** For any ADP \( P_i \) with left end \( x_l \) and right end \( x_r \), extending the part (subpath) cannot decrease the minimax-regret, i.e., \( R_{l_i r_i} \leq R_{l_i r_{i+1}} \) and \( R_{l_i r_i} \leq R_{l_{i-1} r_i} \). In other words, extending a subpath cannot decrease the \( R_{lr} \) term for that subpath. (Proof in Appendix E.1.)

**Lemma 4.** Let \( \hat{P}_q = \{ P_1, P_2, ..., P_q \} \) be the minimax-regret placement of the first \( q \) parts when the right-end of the \( q \)th part is restricted to be \( x_{r_q} \). Let \( \hat{P}'_q = \{ P'_1, P'_2, ..., P'_q \} \) be the minimax-regret placement of the first \( q \) parts when the right-end of the \( q \)th part is restricted to be \( x_{r_q+1} \). Then, if the left end (resp. right end) of the \( i \)th part in \( \hat{P}_q \) is \( x_{l_i} \) (resp. \( x_{r_i} \)) and in \( \hat{P}'_q \) is \( x'_{l_i} \) (resp. \( x'_{r_i} \)), then

\[
\max_{1 \leq i \leq q} \{ R_{l_i r_i} \} \leq \max_{1 \leq i \leq q} \{ R'_{l_i r_i} \}
\]

In other words, the minimax-regret considering only the first \( q \) parts as ADPs for the \( q \)-partition \( \hat{P}'_q \) is greater than or equal to that for the \( q \)-partition \( \hat{P}_q \). (Proof in Appendix E.2.)

### 4.3 Unimodality of Minimax-Regret w.r.t Parts

**Property 3.** In the Minimax-Regret \( k \)-sink Location Problem, the minimax-regret as a function of the left end of the \( i \)th part is unimodal with a unique minimum value given that parts \( \{ P_{i+1}, P_{i+2}, ..., P_k \} \) are fixed. (Proof in Appendix F.2.)

### 5 A Binary Search Based Algorithm

In this section, we give a binary-search based algorithm for finding the Minimax-Regret \( k \)-Sink Locations. From Eq. 12 we know that the \( k \)-sink minimax-regret can be written as:

\[
\min_{\{\hat{P}, \hat{Y}\}} R_{\max}(\{\hat{P}, \hat{Y}\}) = \min_{\hat{P}} \max_{1 \leq i \leq k} \{ R_{l_i r_i} \}
\]
Since we have proved in Property 3 that the minimax-regret is unimodal as a function of the left end of the $i^{th}$ part when the parts $\{P_{i+1}, P_{i+2}, ..., P_k\}$ are fixed, we can do a nested binary search for the locations. First, binary search for the left end of the $k^{th}$ part. For each fixed location, binary search for the left end of the $(k-1)^{st}$ part, and so on until searching for the left end of the 2$^{nd}$ part. This searching will result in $O((\log n)^{k-1})$ possible placements of the parts.

For each fixed placement of the parts, the minimax-regret calculation involves finding the $R_{l,r_i}$ value for all parts $P_i$, and taking the maximum of the values. This can be done in $O(n \log n)$ time. The details of this procedure can be found in Appendix F.1. This procedure to find all $R_{l,r_i}$ values for fixed placements is called at most $O((\log n)^{k-1})$ times. Therefore, the total running time of the algorithm is $O(n^{1+k}(\log n)^{1+\log k})$.

This algorithm performs better than the previously known best result of $O(n^{1+k}(\log n)^{1+\log k})$ for general $k$ given in [11].

6 An Alternative Approach

The binary-search based algorithm, though better than previously known results, is not very asymptotically pleasing since there is a $k$ in the exponent. In this section, we are going to give a different approach to solving the minimax-regret $k$-sink location problem. Using the characterization of the minimax-regret given in Eq. 12, we derive a recurrence for the minimax-regret and implement the recurrence using a Dynamic Programming (DP) approach in Sect. 7.

6.1 A Recurrence for the Minimax-Regret

From Eq. 12, the minimax-regret can be written as:

$$\min_{\tilde{P}, \hat{Y}} R_{\max}(\{\tilde{P}, \hat{Y}\}) = \min_{\tilde{P}} \max_{1 \leq i \leq k} \{R_{l_i,r_i}\}$$

$$= \min_{\tilde{P}} \max_{1 \leq i \leq k-1} \{R_{l_i,r_i}, R_{l_k,r_k}\}$$

$$= \min_{P_k} \max_{0 \leq j \leq k} \{R_{l_i,r_i}, R_{l_k,r_k}\}$$

(13)

The term $\tilde{P} - P_k$ denotes the set $\{P_1, P_2, ..., P_{k-1}\}$. This gives rise to a natural recurrence for the minimax-regret:

$$\min_{\tilde{P}} \max_{0 \leq j \leq k} \{R_{l_i,r_j}\} = \min_{P_k} \max_{0 \leq j \leq k} \{R_{l_i,r_j}, R_{l_k,r_k}\}$$

Let $M(q,i)$ denote the minimax-regret considering only the first $q$ parts as ADPs and the right end of the $q^{th}$ part is $x_i$. From Eq. 13, the following recurrence relation holds for the minimax-regret:

$$M(q,i) = \min_{0 \leq j \leq i} \{\max(M(q-1,j-1), R_{j,i})\}$$

(14)
The objective of the algorithm would be to find $M(k, n)$. In Sect. [7] we implement a DP algorithm based on this recurrence relation.

7 A Dynamic Programming Based Algorithm

In Sect. [6] a recurrence for the minimax-regret was defined in Eq. [14]. Note that in the recurrence, the index $j - 1$ denotes the right end of the $(q - 1)^{th}$ part. We will implement this recurrence by first precomputing $R_{ji}$ for all possible values of $j$ and $i$ ($j < i$). Thus $R_{ji}$ for any $j, i$ will be available in a lookup table accessed in $O(1)$ time. This precomputation is described in Sect. [7.1].

A naive DP table filling implementation of the recurrence in Eq. [14] is going to give us an $O(kn^2)$ procedure. This can be improved to $O(kn)$. However, since the DP procedure is not the bottleneck for the algorithm, we defer the details of the $O(kn)$ improvement to Appendix [G].

7.1 Precomputation of $R_{ji}$ for all subpaths

$R_{ji}$ needs to be precomputed and stored for all subpaths (from $x_j$ to $x_i$) for use in the DP algorithm. Before we precompute $R_{ji}$ for all $j, i$ ($j < i$) pairs, we need to precompute the optimal $k$-Sink evacuation time for all candidate worst-case scenarios $s^*_{Bj} \in S^*$. From Sect. [3] given a scenario we can calculate the $k$-Sink evacuation time in $O(kn \log n)$ time. Also, from Prop. [2], we know that there are a total of $O(n^2)$ candidate worst-case scenarios. Therefore, this precomputation takes time $O(kn^3 \log n)$.

Naive Approach - $O(n^5)$

1. There are $O(n^2)$ possible subpaths (Assumed Dominant Parts), say with left-end $x_l$ and right-end $x_r$.
2. Within each subpath, there are $O(n)$ possible sinks $x_t (l \leq t \leq r)$, and
3. For each sink in the ADP there can be $O(n)$ candidate worst-case scenarios $s^*_{Bj} \in S^*$ (From Theorem [1]).
4. Given an ADP as a subpath from $x_l$ to $x_r$, a sink $x_t$ and a scenario $s^*_{Bj}$ in $O(n)$ time we can calculate the regret $R_{lr}(s^*_{Bj}, x_t) = \Theta^1([x_l, x_r], x_t, s^*_{Bj}) - \Theta^k_{opt}(P, s^*_{Bj})$, since we have already pre-calculated $\Theta^k_{opt}(P, s^*_{Bj})$, the optimal $k$-sink evacuation time for all candidate worst-case scenarios.

Therefore, the total time complexity to precompute $R_{ji}$ using this approach is $O(n^5)$. At this point, we already have a polynomial time algorithm for the Minimax-Regret $k$-Sink Location Problem, with precomputation of $R_{ji}$’s in $O(n^5)$ time dominating the running time (The precomputation of $\Theta^k_{opt}(P, s^*_{Bj})$ for all worst-case scenarios takes $O(kn^3 \log n)$ time and the DP table filling procedure takes $O(kn)$ time.)

However, by observing some properties of the minimax-regret, we are able to bring down the running time of the $R_{ji}$ precomputation to $O(n^3)$. The reduction from $O(n^5)$ to $O(n^3)$ running time is given in Appendix [H].
7.2 Total Running Time

The total running time can be split as:

- Precomputing optimal $k$-sink evacuation times for the $O(n^2)$ candidate worst-case scenarios - $O(kn^3 \log n)$.
- Precomputation of $R_{ji}$'s - $O(n^3)$.
- Algorithm to find minimax-regret $k$-Sink location - $O(kn)$.

The overall running time is dominated by the precomputation of optimal $k$-sink evacuation times for candidate worst-case scenarios. Thus, the running time of the algorithm is $O(kn^3 \log n)$.

References

1. Binay K. Bhattacharya and Tsunehiko Kameda. A linear time algorithm for computing minmax regret 1-median on a tree. In COCOON’2012, pages 1–12, 2012.
2. Sin-Wing Cheng, Yuya Higashikawa, Naoki Katoh, Guanqun Ni, Bing Su, and Yinfeng Xu. Minimax regret 1-sink location problems in dynamic path networks. In Proceedings of TAMC’2013, pages 121–132, 2013.
3. L. R. Ford and D. R. Fulkerson. Constructing Maximal Dynamic Flows from Static Flows. Operations Research, 6(3):419–433, June 1958.
4. Yuya Higashikawa, M. J. Golin, and Naoki Katoh. Minimax Regret Sink Location Problem in Dynamic Tree Networks with Uniform Capacity. In Proceedings of the 8’th International Workshop on Algorithms and Computation (WALCOM’2014), pages 125–137, 2014.
5. Yuya Higashikawa, Mordecai J Golin, and Naoki Katoh. Multiple Sink Location Problems in Dynamic Path Networks. In Proceedings of the 2014 International Conference on Algorithmic Aspects of Information and Management (AAIM 2014), 2014.
6. B Hoppe and É Tardos. The quickest transshipment problem. Mathematics of Operations Research, 25(1):36–62, 2000.
7. Naoyuki Kamiyama, Naoki Katoh, and Atsushi Takizawa. An efficient algorithm for evacuation problems in dynamic network flows with uniform arc capacity. In Proceedings of the Second international conference on Algorithmic Aspects in Information and Management, pages 231–242, 2006.
8. Panos Kouvelis and Gang Yu. Robust Discrete Optimization and Its Applications. Kluwer Academic Publishers, 1997.
9. Hongmei Li, Yinfeng Xu, and Guanqun Ni. Minimax regret vertex 2-sink location problem in dynamic path networks. Journal of Combinatorial Optimization, February 2014.
10. Satoko Mamada, Takeaki Uno, Kazuhisa Makino, and Satoru Fujishige. An $O(n \log^2 n)$ algorithm for the optimal sink location problem in dynamic tree networks. Discrete Applied Mathematics, 154(2387-2401):251–264, 2006.
11. Guanqun Ni, Yinfeng Xu, and Yucheng Dong. Minimax regret $k$-sink location problem in Dynamic Path networks. In Proceedings of the 2014 International Conference on Algorithmic Aspects of Information and Management (AAIM 2014), 2014.
12. Haitao Wang. Minmax Regret 1-Facility Location on Uncertain Path Networks. Proceedings of the 24th International Symposium on Algorithms and Computation (ISAAC’13), pages 733–743, 2013.
Appendix

A An $O(kn \log n)$ Dynamic Programming Algorithm for the Optimal $k$-Sink Location Problem

From this point onwards, whenever we refer to the optimal $k$-sink location problem, we always mean the general discrete version, i.e with the ceiling function kept and any capacity $c > 0$ (not necessarily $c = 1$ in Eqs. 1 and 2.)

We present a solution to the optimal $k$-sink location problem by first establishing a recurrence for the optimal $k$-sink evacuation time and giving a DP algorithm to implement the recurrence. Before doing that, we are going the state the following intuitive lemma with proof deferred to Appendix B.1:

**Lemma 5.** Given a scenario, the optimal $k$-sink evacuation time of a path is greater than or equal to the optimal $k$-sink evacuation time of any of its subpaths.

### A.1 Recurrence for Optimal $k$-Sink Evacuation Time

We are going to present a recurrence for the optimal $k$-sink evacuation time given a scenario $s \in S$. Although we prove the recurrence here, an alternative rigorous derivation based on earlier definitions can be found in Appendix B.2.

Let $T(q, i)$ be the optimal evacuation time considering only the first $q$ parts restricted to the subpath from $x_0$ to $x_i$. Let $w_{ji}$ be the optimal 1-sink evacuation time in the subpath from $x_j$ to $x_i$. Consider the term:

$$\min_{0 \leq j \leq i} \max_{0 \leq l \leq q} (T(q - 1, j - 1), w_{ji})$$

For any particular value of $j$, this formula models the $q$-sink evacuation time restricted to the subpath from $x_0$ to $x_i$, with the left end of the $q$th part being $x_j$, and the first $q - 1$ parts being placed optimally for evacuating the subpath from $x_0$ to $x_{j-1}$. The term is minimized over all possible values of $j$, the left end of the $q$th part.

Since $T(q, i)$ is the optimal solution,

$$T(q, i) \leq \min_{0 \leq j \leq i} \max_{0 \leq l \leq q} (T(q - 1, j - 1), w_{ji}) \quad (15)$$

Under scenario $s \in S$, let us assume that in the optimal $q$-sink placement in the subpath from $x_0$ to $x_i$, the left end of the $q$th part is $x_{j'}$, and the first $q - 1$ parts/sinks are given by $\hat{P}_{q-1} = \{P_1, P_2, ..., P_{q-1}\}$ and $\hat{Y}_{q-1} = \{y_1, y_2, ..., y_{q-1}\}$, the right end of the $(q-1)^{th}$ part being $x'_{j'-1}$. Let $\Theta^{q-1}([x_0, x'_{j'-1}], \{\hat{P}_{q-1}, \hat{Y}_{q-1}\})$ denotes the $(q-1)$-sink evacuation time under scenario $s$ considering the parts/sinks $\hat{P}_{q-1}$ and $\hat{Y}_{q-1}$ in the subpath from $x_0$ to $x_{j'-1}$. Since $T(q - 1, j' - 1)$ is the optimal (minimum) $(q - 1)$-sink evacuation time in the subpath from $x_0$ to $x_{j'-1}$,

$$T(q - 1, j' - 1) \leq \Theta^{q-1}([x_0, x'_{j'-1}], \{\hat{P}_{q-1}, \hat{Y}_{q-1}\}) \quad (16)$$
Now,

\[ T(q, i) = \max \left( \Theta^{q-1}([x_0, x_{j'-1}], \{ \hat{P}_{q-1}, \hat{Y}_{q-1} \}), w_{j'i} \right) \]

\[ \geq \max (T(q - 1, j' - 1), w_{j'i}) \quad \text{(from Eq. 16)} \]

\[ \geq \min_{0 \leq j \leq i} \max (T(q - 1, j - 1), w_{ji}) \] \hspace{1cm} (17)

From Eq. 15 and 17 we see that

\[ T(q, i) = \min_{0 \leq j \leq i} \max (T(q - 1, j - 1), w_{ji}) \] \hspace{1cm} (18)

where \( w_{ji} \) is the optimal 1-sink evacuation time in the subpath from \( x_j \) to \( x_i \).

There can be multiple values of \( j \) (the left end of the \( q^{th} \) part) for which \( T(q, i) \) is minimized. We will always assume without loss of generality that optimal \( j \) is the largest of such values. In other words, the location of the optimal left-end of the \( q^{th} \) part is the rightmost possible.

### A.2 A DP Algorithm for Optimal \( k \)-Sink Location Problem

In this section we provide a DP algorithm to implement the recurrence given in Eq. 18. Note that the DP table for the recurrence will have \( O(kn) \) entries. The row index of the DP table corresponds to \( q \), the number of sinks, and the column index \( i \) corresponds to \( x_i \), the right end-point of the subpath being considered.

We fill the table row-by-row so that when we are calculating \( T(q, i) \), the values of \( T(q - 1, j - 1) \) (for \( 0 \leq j \leq i \)) are already present in the DP table. Again, we note that there can be multiple values of \( j \) that minimize \( T(q, i) \) in Eq. 18 and so, we always choose and refer to the largest possible such value.

We observe two crucial properties of the recurrence which will help us reduce the running time of the DP:

**Property 4.** In the recurrence given by Eq. 18 if we increment \( i \) keeping \( q \) fixed, then the minimizing \( j \) value cannot decrease.

*Proof.* See Appendix B.3

**Property 5.** In the recurrence given by Equation 18 for fixed \( q \) and \( i \), \( \max (T(q - 1, j - 1), w_{ji}) \) is unimodal with a unique minimum value as a function of \( j \).

*Proof.* From Lemma 5 if we increase(resp. decrease) the size of a path, then the optimal \( k \)-sink evacuation time cannot decrease(resp. cannot increase). Thus, when we increase \( j \), \( T(q - 1, j - 1) \) (the optimal \( (q - 1) \)-sink evacuation time on the subpath from \( x_0 \) to \( x_{j-1} \)) cannot decrease and \( w_{ji} \) (the optimal 1-sink evacuation time on the subpath from \( x_j \) to \( x_i \)) cannot increase. Thus, \( \max (T(q - 1, j - 1), w_{ji}) \) will have a unique minimum value as a function of \( j \). \( \square \)
Remark 1. Properties 4 and 5 stated above imply that if we know the minimizing \( j \) value for \( T(q, i - 1) \) (say \( j' \)) then the minimizing \( j \) value for \( T(q, i) \) can be found by scanning linearly from \( j' \), stopping when we find the minimum of the unimodal function \( \max(T(q - 1, j - 1), w_{ji}) \). Note that we will have to overshoot the scanning of \( j \) by 1 node in order to determine the minimum of the unimodal function \( \max(T(q - 1, j - 1), w_{ji}) \).

The recurrence can be re-written as:

\[
T(q, i) = \min_{j' \leq j \leq i} \max(T(q - 1, j - 1), w_{ji}) \tag{19}
\]

where \( j' \) is the minimizing value of \( j \) for \( T(q, i - 1) \).

The recurrence requires the maintenance of \( w_{ji} \), the optimal 1-sink evacuation time in the subpath from \( x_j \) to \( x_i \) as \( i \) and \( j \) increases. In Appendix A.3, we introduce a new data structure to maintain \( w_{ji} \) as \( i \) and \( j \) are incremented. Each increment of \( i \) or \( j \) can be handled in \( O(\log n) \) time.

The DP table filling procedure is given below:

1. Fill in \( T(1, i) = w_{0i} \) (for \( 0 \leq i \leq n \)), the optimal 1-sink evacuation time in the subpath from \( x_0 \) to \( x_i \). (\( O(n \log n) \) time)

2. For \( q \leftarrow 2 \) to \( k \):
   - \( j \leftarrow 0 \)
     - For \( i \leftarrow 0 \) to \( n \):
       - Update \( w_{ji} \)
       - Do \( j \leftarrow j + 1 \) and update \( w_{ji} \) till minimum of \( \max(T(q - 1, j - 1), w_{ji}) \) is found which is equal to \( T(q, i) \). Note that we will have to overshoot the scanning of \( j \) by 1 in order to determine the minima.
       - Store \( T(q, i) \) in the DP table along with the minimizing \( j \) value and the optimal sink for subpath \([x_j, x_i]\) (used to reconstruct the optimal \( k \)-partition and sinks.)

As we can see by the DP procedure, for every \( q \) value, \( i \) is incremented at most \( n \) times and \( j \) at most \( 2n \) times (because of the overshooting by 1). Thus, \( w_{ji} \) is updated atmost \( O(n) \) times for every \( q \) value and every update incurs a cost of \( O(\log n) \) to maintain \( w_{ji} \). Thus, we are able to fill the \( O(kn) \) DP table entries in \( O(kn \log n) \) time.

A.3 Maintenance of \( w_{ji} \)

In the DP algorithm, we also need to maintain \( w_{ji} \), the optimal 1-sink evacuation time, as \( j \) and \( i \) increases. Therefore any data structure which maintains \( w_{ji} \) must be able to maintain the maximum of the evacuation function of every node defined in the Eqs. 4 and 5 and support the following operations:

1. Return the optimal 1-sink evacuation time in subpath \( \{x_j, ..., x_i\} \).
2. When \( i \leftarrow i + 1 \), then append a new node to the right of the path.
3. When \( j \leftarrow j + 1 \), then delete the leftmost node from the path.

We introduce a new data structure which we call a Bi-Heap which can maintain \( w_{ji} \) and perform all operations in \( O(\log n) \) time. For details of the data structure, refer Appendix C.
A.4 Running time

We are filling $O(kn)$ DP entries and each takes $O(\log n)$ time (for maintaining $w_{ji}$). Thus the overall running time of the algorithm is $O(kn \log n)$.

B Proofs for the Optimal $k$-Sink Evacuation

B.1 Proof of Lemma 5

Assume a path $P$ and a subpath $P'$ of the path $P$. Now, given a scenario, if the optimal $k$-sink evacuation time in $P'$ is greater than the optimal $k$-sink evacuation time in $P$, then we can use the $k$-partition used in the optimal solution for $P$ on $P'$ ($\leq k$ parts) and obtain a better solution than the optimal one in $P'$. Therefore, the optimal $k$-sink evacuation time in $P$ is greater than or equal to the optimal $k$-sink evacuation time in any of its subpaths $P'$.

B.2 A Derivation for the Optimal $k$-Sink Evacuation Recurrence

The $k$-sink evacuation time is given by the following equation:

$$\Theta^k(P, \{ \hat{P}, \hat{Y} \}, s) = \max_{1 \leq i \leq k} \Theta^1(P_i, y_i, s)$$

(from Eq. 6)

$$= \max \left( \max_{1 \leq i \leq k-1} \{ \Theta^1(P_i, y_i, s) \}, \Theta^1(P_k, y_k, s) \right)$$

(20)

$$= \max \left( \Theta^{k-1}(P - P_k, \{ \hat{P} - P_k, \hat{Y} - y_k \}, s), \Theta^1(P_k, y_k, s) \right)$$

(21)

where $P - P_k$ denotes the path excluding the $k^{th}$ part and $\hat{P} - P_k$ denotes the set $\{ P_1, P_2, ..., P_{k-1} \}$.

Thus the optimal $k$-sink evacuation time can be written as:

$$\Theta^k_{opt}(P, s) = \min_{\{ \hat{P}, \hat{Y} \}} \Theta^k(P, \{ \hat{P}, \hat{Y} \}, s)$$

(22)

$$= \min_{\{ \hat{P}, \hat{Y} \}} \max \left( \Theta^{k-1}_{opt}(P - P_k, s), \Theta^1(P_k, y_k, s) \right)$$

As we can see from Eq. 22, a natural recurrence relation exists for the optimal $k$-sink evacuation time.
Let $T(q, i)$ be evacuation time considering only the first $q$ parts and only in the subpath from $x_0$ to $x_i$. From Eq. 22, we can write $T(q, i)$ as:

$$T(q, i) = \min_{0 \leq j \leq i} \max (T(q - 1, j - 1), w_{ji})$$

where $w_{ji}$ is the optimal 1-sink evacuation time in the subpath from $x_j$ to $x_i$. There can be multiple values of $j$ (the left end of the last part) for which $T(q, i)$ is minimized. We assume w.l.g that the largest value of minimizing $j$ is always chosen and meant when we say minimizing $j$ value.

### B.3 Proof of Property 4

Equation 18 is given by:

$$T(q, i) = \min_{0 \leq j \leq i} \max (T(q - 1, j - 1), w_{ji})$$  (23)

where $w_{ji}$ is the optimal 1-sink evacuation time in the subpath from $x_j$ to $x_i$.

Assume $j' \prime$ is the minimizing $j$ value for $T(q, i - 1)$ and $j''$ is the minimizing $j$ value for $T(q, i)$. We need to prove that $j'' \geq j'$.

Assume the contrary, i.e., $j'' < j'$. We now have two cases to deal with:

**Case 1:** $w_{j'(i-1)} \geq T(q - 1, j' - 1)$

In this case, the $q$-sink evacuation time in the subpath from $x_0$ to $x_{i-1}$ will be $T(q, i - 1) = w_{j'(i-1)}$. Now,

$$T(q, i) = \max (T(q - 1, j'' - 1), w_{j''i})$$

**Remark 2.** The optimal $k$-sink evacuation time of a path is greater than or equal to the $k$-sink evacuation time of any of its subpaths. (From Lemma 5)

Since $j'' < j'$,

$$T(q - 1, j'' - 1) \leq T(q - 1, j' - 1)$$  (from Remark 2)

$$\leq w_{j'(i-1)}$$  (from case assumption)

$$\leq w_{j'i}$$  (from Remark 2)

$$\leq w_{j''i}$$  (from Remark 2)  (24)

Therefore from Eq. 24, $T(q, i) = w_{j''i}$. Since $j''$ is the largest minimizing $j$ value for $T(q, i)$ and $j' > j''$,

$$w_{j''i} < \max (T(q - 1, j' - 1), w_{j'i})$$  (25)

$$= w_{j'i}$$  (26)

But from Remark 2 we know that the optimal 1-sink evacuation time of the path $x_{j''}...x_i$ cannot be strictly lesser than the 1-sink evacuation time of one of its subpaths $x_{j'}...x_i$. Eq. 25 cannot be true. We arrive at a contradiction. Therefore, our assumption that $j'' < j'$ is false for this case. So, $j'' \geq j'$.

17
**Case 2:** \( w_{j'(i-1)} < T(q - 1, j' - 1) \)

In this case, the \( q \)-sink evacuation time in the subpath from \( x_0 \) to \( x_{i-1} \) will be \( T(q, i-1) = T(q - 1, j' - 1) \). The minimizing \( j \) value for \( T(q, i) \) is \( j'' \),

\[
T(q, i) = \max (T(q - 1, j'' - 1), w_{j'i})
\]

Since \( j'' \) is the largest minimizing \( j \) value for \( T(q, i) \), and \( j' > j'' \),

\[
\max (T(q - 1, j'' - 1), w_{j'i}) < \max (T(q - 1, j' - 1), w_{j'i}) \quad (27)
\]

By the case assumption and the fact that \( j' \) is the minimizing \( j \) value for \( T(q, i - 1) \),

\[
w_{j'(i-1)} < T(q - 1, j' - 1) \quad \text{(from case assumption)}
\]

\[
\leq \max (T(q - 1, j'' - 1), w_{j''(i-1)})
\]

\[
\leq \max (T(q - 1, j'' - 1), w_{j''i}) \quad \text{(from Remark 2)} \quad (28)
\]

Therefore,

\[
T(q - 1, j' - 1) \leq \max (T(q - 1, j'' - 1), w_{j''i}) \quad \text{(from Eq. 28)} \quad (29)
\]

\[
w_{j'i} \leq w_{j''i} \quad \text{(from Remark 2)} \quad (30)
\]

From Eqs. [29 and 30]

\[
\max (T(q - 1, j' - 1), w_{j'i}) \leq \max (T(q - 1, j'' - 1), w_{j''i}) \quad (31)
\]

We see that Eqs. [27 and 31] contradict each other. Therefore, our assumption that \( j'' < j' \) is false for this case. So, \( j'' \geq j' \). \( \square \)

**C The Bi - Heap Data Structure**

We introduce this data structure to help maintain \( w_{ji} \), the optimal 1-sink evacuation time in the subpath from \( x_j \) to \( x_i \) as \( j \) and \( i \) are incremented in \( O(\log n) \) time. This maintenance can be done efficiently by combining together various basic data structures, e.g., heaps and 2-3 trees. While not difficult, the details are quite technical.

**Remark 3.** In the subpath from \( x_j \) to \( x_i \), the position of the optimal 1-sink cannot move left if \( j \) and/or \( i \) are incremented.

**C.1 Setup**

For any node \( x_i (0 \leq i < n) \), set \( \ell_i = x_{i+1} - x_i \) to be the distance from node \( i \) to \( i + 1 \) and \( w_i > 0 \) be the number of people at node \( i \) who need to be evacuated.

Let \( \ell_{i,j} = x_j - x_i \) and \( W_{i,j} = \sum_{i \leq t \leq j} w_t \). Suppose all edges have fixed uniform capacity \( c \). For \( i < j \) the time to evacuate the nodes in \( [i, j] \) to node \( j \) is \( E_{i,j} - 1 \) and the time to evacuate the nodes in \( [i, j] \) to node \( j \) is \( E'_{i,j} - 1 \) where

18
\[ E_{i,j} = \max_{i \leq t < j} \left( \left\lceil \frac{W_{i,t}}{c} \right\rceil + \ell_{t,j} \right), \quad E'_{i,j} = \max_{i < t \leq j} \left( \left\lceil \frac{W_{i,j}}{c} \right\rceil + \ell_{i,t} \right) \] (32)

In order to maintain the optimal 1-sink evacuation time for the last part, we maintain the left evacuation time \( E_{i,j} \) and the right evacuation time \( E'_{i,j} \) for the optimal sink. Note that by Remark 3, the optimal sink in the last part cannot move to the left. This implies that we must be able to handle the updates from \( E_{i,j} \) to \( E_{i,j+1} \) or \( E_{i,j+1} \) and from \( E'_{i,j} \) to \( E'_{i+1,j} \) or \( E'_{i,j+1} \).

One method of handling these updates is doing a \( O(n^2) \) preprocessing step that precalculates all possible \( E_{i,j} \) and \( E'_{i,j} \) values for later use, using \( O(n^2) \) space.

Another approach, the one that will be outlined here, is to create a \( O(n) \) size data structure that permits reusing old information about \( E_{i,j} \) or \( E'_{i,j} \) to calculate the value of \( E_{i+1,j} \), \( E_{i,j+1} \), \( E'_{i+1,j} \), or \( E'_{i,j+1} \) in \( O(\log n) \) time.

### C.2 Some observations

Suppose \( i < j \) are given and \( j \) is increased by 1. Note that

\[
E_{i,j+1} = \max_{i \leq t < j+1} \left( \left\lceil \frac{W_{i,t}}{c} \right\rceil + \ell_{t,j+1} \right)
= \max \left( \max_{i \leq t < j} \left( \left\lceil \frac{W_{i,t}}{c} \right\rceil + \ell_{t,j} \right) , \left\lceil \frac{w_{j+1}}{c} \right\rceil + \ell_{j} \right)
\]

That is, the maximum is taken over *almost* the same set of items as \( E_{i,j} \) except that \( \ell_j \) is added to all of the old items and one new item is added.

If \( i \) is increased by one then

\[
E_{i+1,j} = \max_{i+1 \leq t < j} \left( \left\lceil \frac{W_{i+1,t}}{c} \right\rceil + \ell_{t,j} \right)
= \max \left( \max_{i+1 \leq t < j} \left( \left\lceil \frac{W_{i,t} - w_{i}}{c} \right\rceil + \ell_{t,j} \right) , \left\lceil \frac{w_{j+1}}{c} \right\rceil + \ell_{i,j+1} \right)
\]

The maximum here is again taken over *almost* the same set of items as \( E_{i,j} \) except that one item is removed from the set and \( w_i \) is subtracted from all of the \( W_t \).

We now examine \( E'_{i,j} \) and find that the situation is very similar.

If \( j \) is increased by 1 then

\[
E'_{i,j+1} = \max_{i \leq t \leq j+1} \left( \left\lceil \frac{W_{t,j+1}}{c} \right\rceil + \ell_{t,j} \right)
= \max \left( \max_{i \leq t \leq j} \left( \left\lceil \frac{W_{t,j} + w_{j+1}}{c} \right\rceil + \ell_{t,i} \right) , \left\lceil \frac{w_{j+1}}{c} \right\rceil + \ell_{i,j+1} \right)
\]
The maximum is taken over *almost* the same set of items as $E'_{i,j}$ except that $w_{j+1}$ is added to all of the old items and one new item is added.

If $i$ increases by 1 we get

$$E'_{i+1,j} = \max_{i+1 < t \leq j} \left( \left\lceil \frac{W_{i,j}}{c} \right\rceil + \ell_{i+1,t} \right)$$

where the maximum is now taken over *almost* the same set of items as $E'_{i,j}$ except that one item is removed from the set and $\ell_{i+1}$ is subtracted from all of the $\ell_{i,t}$.

### C.3 The Bi-Heap Data Structure

In order to simplify the exposition we define.

$$\text{Cost}(W, L) = \left\lceil \frac{W}{c} \right\rceil + L.$$  

Following the observations in the previous section we note that to solve the problem it suffices to implement a data structure that maintains a set of pairs $Z = \{(W_i, L_i)\}$, allowing the following operations on the set, with each operation requiring only $O(\log n)$ time:

1. MAX: returns $\max \{\text{Cost}(W_i, L_i) : (W_i, L_i) \in Z\}$
2. AddW$(w)$: In every $(W_i, L_i) \in Z$, replace $W_i$ by $W_i + w$.
3. AddL$(\ell)$: In every $(W_i, L_i) \in Z$, replace $L_i$ by $L_i + \ell$.
4. Insert$(W, L)$: Insert new pair $(W, L)$ into $Z$
5. Delete$(W, L)$: Remove pair $(W, L)$ from $Z$. Note that the input here will be a pointer to the current location of $(W, L)$ in $Z$.

We will call such a data structure a Bi-Heap. $E_{i,j}$ can be evaluated by maintaining a Bi-Heap on the appropriate $(W, L)$ pairs and calculating $E_{i,j}$ via the MAX function. $E_{i,j}$ can be updated to $E'_{i,j}$ in $O(\log n)$ time by first performing a AddL$(\ell_j)$ operation and then an Insert$((w_j, \ell_j))$ one. The value of $E_{i,j+1}$ is then found by calling MAX. A similar observation permits updating in $O(\log n)$ time from $E_{i,j}$ to $E'_{i,j}$, and from $E'_{i,j}$ to $E''_{i,j}$ or $E'_{i+1,j}$.

We will now see how to implement such a data structure.

At time $t$ define $\tilde{w}_t = w$ if the current operation is AddW$(w)$; define $\tilde{\ell}_t = \ell$ if the current operation is AddL$(\ell)$. Otherwise, set $\tilde{w}_t = 0$ and $\tilde{\ell}_t = 0$. Define $\tilde{W}_t = \sum_{t' \leq t} \tilde{w}_{t'}$ and $\tilde{L}_t = \sum_{t' \leq t} \tilde{\ell}_{t'}$. Our algorithm will keep the current value of $\tilde{W}_t$ in a variable $\tilde{W}$ and the current value of $\tilde{L}_t$ in a variable $\tilde{L}$. This can be maintained in $O(1)$ time per step.

The main issue in designing the data structure is dealing with the $\left\lceil \frac{W}{c} \right\rceil$ term in the cost function. Observe that $\text{Cost}(W_i, L_i + \ell) = \text{Cost}(W_i, L_i) + \ell$. Thus AddL$(\ell)$ does not change the relative ordering of the items in $Z$. If $\text{Cost}(W, L)$
was actually defined by $\frac{W}{c} + L$ then $\text{Cost}(W_i + w, L) = \text{Cost}(W_i, L_i) + \frac{w}{c}$ so $\text{AddW}(w)$ would also not change the relative ordering of the items in $Z$. This means that $Z$ could be maintained by a priority queue in $O(\log n)$ time per operation. The only subtle point is that the priority queue would not store the actual current values $(W_i, L_i)$ in the data structure. Let $t'$ be the time at which item $(W, L)$ was inserted. The priority queue will store $W, L$ and the associated values $\bar{W}, \bar{L}$. The value of the pair in the priority queue at the current time is

$$(W + W - \bar{W}, L + L - \bar{L})$$

which can be calculated on the fly in $O(1)$ time.

The reason this idea does not fully work for the actual problem is that with the real definition $\text{Cost}(W, L) = \lceil \frac{W}{c} \rceil + L$ the relative ordering of the cost function is not maintained under $\text{AddW}(w)$. More specifically, it is not difficult to find examples when $\text{Cost}(W, L) < \text{Cost}(W', L')$ but $\text{Cost}(W + w, L) > \text{Cost}(W' + w, L')$. So, $\text{AddW}(w)$ operations can change the relative ordering of the items in $Z$ under $\text{Cost}$. It is this complication that requires augmenting the data structure.

This issue can be addressed by first partitioning the $(W_i, L_i)$ pairs into (at most) $c$ subsets $Z_d, 0 \leq d < c$ where

$$Z_d = \{(W_i, L_i) \in Z : W_i \equiv d \mod c\}$$

Note that

$$\max \{\text{Cost}(W_i, L_i) : (W_i, L_i) \in Z\} = \max_{0 \leq d < c} R_d$$

where

$$R_d = \{\text{Cost}(W_i, L_i) : (W_i, L_i) \in Z_d\}.$$ 

The idea is, to identify, for each $Z_d$, the item with largest cost, $R_d$, and only maintain the maximum over those representative values and not over the full set of items. These $R_d$ will be kept in an auxiliary max-Heap $H$.

Next note that, if $W \equiv W' \mod c$, then

$$\left\lfloor \frac{W + w}{c} \right\rfloor - \left\lfloor \frac{W}{c} \right\rfloor = \left\lfloor \frac{W' + w}{c} \right\rfloor - \left\lfloor \frac{W'}{c} \right\rfloor.$$ 

So, by definition, for fixed $d$, the items in a fixed $Z_d$ stay in the same cost sorted order after $\text{AddW}$ and $\text{AddL}$ operations. Thus, the representative of $Z_d$ remains the same after those operations.

Our algorithm will keep each set $Z_d$ in its own max-heap. Note that there is some ambiguity in the labelling of $Z_d$. Suppose that $(W, L) \in Z_d$. After an $\text{AddW}(w)$ operation $W$ is replaced by $W + w$ so the pair is now in $Z_{d'}$ where

$$d' = (W + w) \mod c = (d + w) \mod c.$$
We therefore label the Max-Heaps by the \( d \) values they would have at time \( t = 0 \). More specifically, we define \( H_0, H_1, \ldots, H_{c-1} \), only explicitly storing the non-empty ones. At time \( t \), \( Z_d \) will be stored in \( H_{d'} \) where \( d' + \bar{W}_t \equiv d \mod c \).

We now define the pieces of our data structure

1. \( \bar{W}, \bar{L} \). These values are set to the current values \( \bar{W} = \bar{W}_t \), and \( \bar{L} = \bar{L}_t \):
   - Implemented using a binary-tree heap data structure ordered using cost \( [W_i/c] + L_i \). Allows identifying the max value, insertion and deletion of an element in \( O(\log n) \) time, where \( n \) is the number of elements in the heap.
   - Only non-empty \( Z_d \) would be stored with one max-heap for each nonempty \( Z_d \)
   - \( Z_d \) is stored in Heap \( H_{d'} \) where \( d' = (d - \bar{W}_t) \mod c \).
   - Each entry in a \( H_{d'} \) contains the \((W, L)\) values of the pair at the time of insertion as well as the values \( \bar{W}_t, \bar{L}_t \) at the time of insertion.
   - Identifies the largest cost item in \( Z_d \) as \( R_d \). This will also be labelled as \( r_{d'} \)

2. A dictionary \( D \) allowing access to the existing \( H_{d'} \) heaps by their labels.
   - Implemented using a binary tree

3. A special type of max Heap \( H \) that stores the (maximum cost) representative from each non-empty \( Z_d \).
   - Stores the representatives, \( r_{d'} \) of the non-empty \( H_{d'} \) heaps
   - Representatives \( r_{d'} \) are stored with their index \( d' \) and a pointer to the entry they represent in \( H_{d'} \).
   - Permits calculating \( \text{MAX} \) (of the representatives) in \( O(1) \) time and inserting and deleting items from \( H \) in \( O(\log n) \) time, where \( n \) is the current number of items in \( H \).
   - Has one more special operation to be described below.

\( H \) will be implemented by using a 2-3 tree with the leaves of the tree being the \( r_{d'} \). In a 2-3 tree, all leaves are on the same depth so the height of the tree is \( O(\log n) \).

The actual cost of a leaf is not stored (since it is always changing) but can be calculated in \( O(1) \) time from its index \( d' \).

The leaves of \( H \) are sorted from left to right by increasing index \( d' \). The leaf descendants of any internal node \( v \) therefore form a continuous interval of leaf nodes. We let \( I(v) \) denote this interval.

Internal node \( v \) contain two pieces of information in addition to pointers to its (2 or 3) children and its parent: the left endpoint which is the smallest index of a representative in \( I(v) \) and the max cost index which is the index of the largest cost item in \( I(v) \). The information in an internal node can therefore be calculated from the information in its (2 or 3) children; the left endpoint is the smallest of the left endpoints of its (2 or 3) children and the max cost node is the max cost index of its (2 or 3) children that has the largest cost. Note that the max cost index in the root \( r \) is the max cost node in \( I(r) \) and is therefore the max cost item in \( Z \).
Considering \( H \) as a max heap on its leaves, standard techniques permit inserting a new leaf or deleting an old one in \( O(n) \) time. Furthermore, note that if all the values of all of the leaves in the tree are changed by the same amount \( \ell \) then the tree does not change at all.

We can now complete the description of the Bi-Heap and its operations.

**Max**: Simply report the max cost node in \( H \).

**Insert** or **Delete**:

Before doing anything else first calculate \( d = (W \mod C) \) to identify the \( Z_d \) to which \((W,L)\) belongs and \( d' = (d - \bar{W}) \mod c \) to identify the \( H_{d'} \) which stores \( Z_{d'} \).

For **Insert**, if \( Z_d \) is empty and no such \( H_{d'} \) exists, create it and insert \( H_{d'} \) into \( D \).

Now insert \((W,L)\) into \( H_{d'} \). If this new item is *not* the largest cost item in \( H_{d'} \), stop. If it is, then delete the current \( d' \)-representative from \( H \) and **Insert** \((W,L)\) into \( H \) as the newest \( d' \)-representative.

For **Delete** first check if \((W,L)\) is the largest item in \( H_{d'} \). If it is not, then just delete it from \( H_{d'} \) and stop. If \( H_{d'} \) is now empty, delete it from \( D \) If \((W,L)\) is the largest item in \( H_{d'} \) then it is also the \( d' \)-representative in \( H \). Delete it from \( H \) and \( H_{d'} \). If \( H_{d'} \) is now empty, delete it from \( D \) If \( H_{d'} \) is not empty then insert the largest remaining item in \( H_{d'} \) into \( H \) as the \( d' \)-representative.

**AddL** (\( \ell \)):

**AddL** (\( \ell \)) operations only change the actual values of the items but not their relative orderings. So, nothing needs to be changed. The change in \( \ell \) value is taken care of by setting \( \bar{L} = \bar{L} + \ell \).

**AddW** (\( w \)):

First set \( \bar{W} = \bar{W} + w \).

We assume \( w > 0 \) (\( w < 0 \) is very similar). Note that for each fixed \( d \), the ordering of the items in \( Z_d \) remains invariant after the operation, so the \( H_{d'} \) structures do not change. The only thing that might change is the relative orderings of the \( r'_{d'} \)-representatives in \( H \).

First assume \( w \equiv 0 \mod c \). Then, for all \((W,L)\), \( \text{Cost}(W+w,L) = \text{Cost}(W,L) + w/c \). Thus the relative ordering of the \( d' \) representatives in \( H \) remains invariant and nothing further needs to be done.

Otherwise, \( w = \bar{k}c + \bar{d} \) where \( \bar{k}, \bar{d} \) are integers and \( \bar{d} > 0 \). Thus \( \lceil w/c \rceil = \bar{k} + 1 \).

Now let \((W,L)\) \( \in Z_d \). Then \( W = kc + d \) and

\[
\text{Cost}(W,L) = \begin{cases} 
  k + L, & \text{if } d = 0 \\
  k + L + 1, & \text{if } d \neq 0 
\end{cases}.
\]
Thus, if $W \in Z_d$, then

$$\text{Cost}(W + w, L) = \left\lceil \frac{k + \bar{k}c + d + \bar{d}}{c} \right\rceil + L$$

(33)

$$= \begin{cases} k + L + \bar{k} + 1 & \text{if } d + \bar{d} \leq c \\ k + L + 2 & \text{if } d + \bar{d} > c \end{cases}$$

(34)

$$= \begin{cases} \text{Cost}(W, L) + \lceil w/c \rceil, & \text{if } d = 0 \\ \text{Cost}(W, L) + \lceil w/c \rceil - 1, & \text{if } d \neq 0, d + \bar{d} \leq c \\ \text{Cost}(W, L) + \lceil w/c \rceil, & \text{if } d + \bar{d} > c \end{cases}$$

(35)

$$= \begin{cases} \text{Cost}(W, L) + \lceil w/c \rceil - 1, & \text{if } 0 < d \leq c - \bar{d} \\ \text{Cost}(W, L) + \lceil w/c \rceil, & \text{otherwise} \end{cases}$$

(36)

For algorithmic purposes we need to transform $d$ into $d'$. Recall that $d = d' + \bar{W}_i \mod c$. Set $x = \bar{W}_i \mod c$. Then $d = d' + x \mod c$.

There are three cases. The first is that $x = 0$. In this case $d'$ and (36) stays the same after substituting $d'$ for $d$.

In the second $c - \bar{d} \leq x < c$ or equivalently, $0 < c - x \leq \bar{d}$, (36) can now be rewritten as

$$\text{Cost}(W + w, L) = \begin{cases} \text{Cost}(W, L) + \lceil w/c \rceil - 1, & \text{if } c - x < d' \leq c - \bar{d} + c - x \\ \text{Cost}(W, L) + \lceil w/c \rceil, & \text{otherwise} \end{cases}$$

(37)

The final case is $0 < x < c - \bar{d}$, for which (36) can be rewritten as

$$\text{Cost}(W + w, L) = \begin{cases} \text{Cost}(W, L) + \lceil w/c \rceil, & \text{if } c - \bar{d} - x < d' \leq c - x \\ \text{Cost}(W, L) + \lceil w/c \rceil - 1, & \text{otherwise} \end{cases}$$

(38)

Now define $I_x \subseteq [0, c)$ as follows:

$$I_x = \begin{cases} (0, c - \bar{d}) & \text{if } x = 0 \\ (c - \bar{d} - x, c - x) & \text{if } 0 < x < c - \bar{d} \\ (c - x, c - \bar{d} + c - x) & \text{if } c - \bar{d} \leq x < c \end{cases}$$

(39)

Suppose that $r_i$ and $r_j'$ are any two $r_{d'}$ representatives in $H$ and an $AddW(w)$ operation has just been performed.

If $w \mod c = 0$ then the relative ordering of $r_i$ and $r_j'$ does not change so $H$ remains unchanged and nothing further needs to be done.

Otherwise, let $x = \bar{W}_i \mod c$ after the $AddW(w)$ operation. If $r_i$ and $r_j'$ are either both in $I_x$ or both outside $I_x$ then they both get changed by the same amount and their relative ordering remains unchanged.

In $H$ this means the max cost index of internal node $v$ can only change if there are two leaf nodes in $I(v)$ such that one of the nodes is in $I_x$ and one outside of $I_x$, i.e., $I_x \cap I(v) \neq \emptyset$. 

24
We will now see that the number of such \( v \) is \( O(\log n) \) and their max cost indices can be updated in \( O(\log n) \) total time.

Set \( i \) be the smallest leaf index in \( H \) in \( I_x \) and \( j \) the largest leaf index in \( H \) in \( I_x \) and set \( I = [i, j] \). Note that \( i, j \) can be found in \( O(\log n) \) time using tree traversal. By definition \( I_x \cap I(v) = I \cap I(v) \) so we will restrict ourselves to examining \( I \cap I(v) \). Let \( h \) be the lowest common ancestor of \( i', j' \) in \( H \) and \( r \) the root of \( H \).

By the construction of the tree it is easy to see that if \( I(v) \cap I(v') \neq \emptyset \) then \( v \) must be on (a) the path from \( i \) to \( h \), (b) the path from \( j \) to \( h \) or the path from \( h \) to \( r \). Since the tree has height \( O(\log n) \) these paths together only contain \( O(\log n) \) vertices.

The algorithm is then to (a) walk up the path from \( i' \) to \( h \) changing the max-cost values as appropriate (this can be done in \( O(1) \) time per step since a node only needs to examine its (2 or 3) children, (b) similarly walk up the path from \( j' \) to \( h \) and then (c) from \( h \) to \( r \). In total, this requires only \( O(\log n) \) time.

After these walks are finished all internal nodes contain the correct max-cost values, so the algorithm is done.

D Proofs for Structure of Worst-Case Scenarios

D.1 Proof of Lemma 1

The lemma states if \( s^*_B \in S \) is transformed such that \( w_i(s^*_B) = w_i^- \) if \( x_i \notin P_d \), then it remains a worst-case scenario. We prove this lemma by first assuming a worst-case scenario \( s^* \in S \) and showing a transformation to \( s^*_B \) with the required structure.

Consider a worst-case scenario \( s^* \in S \) with dominant part \( P_d \). Now construct a new scenario \( s^*_B \) from \( s^* \) by making the following changes:

1. \( w_i(s^*_B) = w_i(s^*) \) for \( x_i \in P_d \), and
2. \( w_i(s^*_B) = w_i^- \) for \( x_i \notin P_d \).

By making this change, the evacuation time for algorithm \( A \)'s choice of \( \{\hat{P}, \hat{Y}\} \) does not change since the scenario within the dominant part \( P_d \) is unchanged. Thus,

\[
\Theta^k(P, \{\hat{P}, \hat{Y}\}, s^*_B) = \Theta^k(P, \{\hat{P}, \hat{Y}\}, s^*) \quad (40)
\]

Since by making this change, we are only reducing the number of people in the buildings, the optimal evacuation time can only decrease. Thus,

\[
\Theta^k_{opt}(P, s^*_B) \leq \Theta^k_{opt}(P, s^*) \quad (41)
\]

By subtracting equations (40) and (41), we can see that the regret under scenario \( s^*_B \) cannot be lesser than in \( s^* \). This implies that \( s^*_B \) is also a worst-case scenario with dominant part \( P_d \in P \) such that \( w_i(s^*_B) = w_i^- \) if \( x_i \notin P_d \).
D.2 Proof of Lemma 2

Consider any arbitrary worst-case scenario \( s^* \in \mathcal{S} \) with dominant part \( P_d \in \hat{P} \). We will show a construction from \( s^* \) to \( s^*_B \in \mathcal{S} \) such that the sub-scenario within \( P_d \) lies in \( \mathcal{S}^d_L \cup \mathcal{S}^d_R \).

The sink chosen by algorithm \( A \) in part \( P_d \) is \( y_d \). Let \( x_l \) (resp. \( x_r \)) be the leftmost (resp. rightmost) node in \( P_d \). Without loss of generality, let us assume that \( \Theta_L(P_d, y_d, s^*) > \Theta_R(P_d, y_d, s^*) \), i.e., the left evacuation time dominates in \( P_d \) for sink \( y_d \) under worst-case scenario \( s^* \). By the equation defined by Eq. 3,

\[
\Theta_L(P_d, y_d, s^*) = \max_{l_d \leq i \leq r_d} \left\{ (x - y_d) \tau + \sum_{l_d \leq j \leq i} w_j(s^*) \mid y_d > x_i \right\}
\]

let \( m \) be the index of the node which maximizes the above term.

We will now show the transformation from \( s^* \) to \( s^*_B \). Consider a node \( x_t \). If \( x_t \in \{x_{m+1}, ..., x_r\} \), then by making \( w_t(s^*_B) = w_t^* \),

\[
\Theta_L(P_d, y_d, s^*_B) = \Theta_L(P_d, y_d, s^*)
\]

\[
\Theta^{k}_{\text{opt}}(P, s^*_B) \leq \Theta^{k}_{\text{opt}}(P, s^*)
\]

In other words, the evacuation time is unchanged and the optimal \( k \)-sink evacuation time cannot increase. Thus, the regret in \( s^*_B \) cannot be greater than the regret in \( s^* \) after this transformation.

If \( x_t \in \{x_1, ..., x_m\} \), then by making \( w_t(s^*_B) = w_t^+ \) (See Fig. 3),

\[
\Theta_L(P_d, y_d, s^*_B) = \Theta_L(P_d, y_d, s^*) + (w_t^+ - w_t(s^*))
\]

\[
\Theta^{k}_{\text{opt}}(P, s^*_B) \leq \Theta^{k}_{\text{opt}}(P, s^*) + (w_t^+ - w_t(s^*))
\]

Thus, again by this transformation the regret in \( s^*_B \) cannot be greater than the regret in \( s^* \). By these two transformations, we have constructed a left-dominant sub-scenario within \( P_d \). A similar proof exists for constructing a right-dominant sub-scenario when the right evacuation time dominates.

Therefore, the sub-scenario in \( P_d \) lies in the set \( \mathcal{S}^d_L \cup \mathcal{S}^d_R \). \( \square \)

D.3 Proof of Property 1

We are given the \( k \)-partition \( \hat{P} = \{P_1, P_2, ..., P_k\} \) by algorithm \( A \). If we know the dominant part \( P_d \in \hat{P} \), then Thm. 1 established the structure of candidate worst-case scenarios.
Let $\ell_i$ be the number of nodes in part $P_i \in \mathcal{P}$. It is easy to see that $\sum_{1 \leq i \leq k}(\ell_i) = n$. If $P_0$ were the dominant part, then there are a possible $O(t_d)$ worst-case scenarios (from Thm. 1). Thus, the total number of candidate worst-case scenarios (assuming each $P_i$ to be the dominant part) is $O(\sum_{1 \leq i \leq k}(\ell_i))$ which is $O(n)$. □

D.4 Proof of Property 2

Irrespective of the choice of $\{\hat{P}, \hat{Y}\}$ by algorithm $A$, the worst-case scenario chosen by adversary $B$, $s^*_B \in \mathcal{S}^*$, is always of the form described in Thm. 1. Looked at from the perspective of path $P$, for some pair of integers $(t_1, t_2)$ such that $(0 \leq t_1 \leq t_2 \leq n + 1)$, $s^*_B$ can be broken down as:

$$s^*_B = \begin{cases} w^-, & 0 \leq i < t_1 \\ w^+, & t_1 \leq i < t_2 \\ w^-, & t_2 \leq i \leq n \end{cases}$$

There are $O(n^2)$ possible values for $(t_1, t_2)$. Therefore, there are $O(n^2)$ possible worst-case scenarios, i.e., $|\mathcal{S}^*| = O(n^2)$.

E Proofs for the Characterization of Minimax-Regret

E.1 Proof of Lemma 3

Consider the part $P_i$ with left end (resp. right end) as vertex $x_l$ (resp. $x_r$). Let $P'_i$ be the part with one node appended to the right of part $P_i$, its left end is $x_l$ and its right end is $x_{r+1}$. From Eq. 11 the minimax-regret for ADP(Assumed Dominant Part) $P_i$ and ADP $P'_i$ can be written as:

$$R_{l, r_i} = \min_{l \leq i \leq r_i} \{R_{l, r_i}(x_l)\}$$

$$R_{l, (r_i+1)} = \min_{l \leq i \leq r_i+1} \{R_{l, (r_i+1)}(x_l)\}$$

We associate $R_{l, r_i}$ with a sequence of terms $Z = (R_{l, r_i}(x_l), R_{l, r_i}(x_{l+1}), ..., R_{l, r_i}(x_r))$ where $R_{l, r_i}$ is the minimum of the terms in the sequence $Z$. Similarly, we associate $R_{l, (r_i+1)}$ with sequence $Z' = (R_{l, (r_i+1)}(x_l), R_{l, (r_i+1)}(x_{l+1}), ..., R_{l, (r_i+1)}(x_{r+1}))$. Note that $Z'$ has one more term than $Z$, which is $R_{l, (r_i+1)}(x_{r+1})$.

Claim. For subpaths $P_i$ and $P'_i$, $R_{l, r_i}(x_t) \leq R_{l, (r_i+1)}(x_t)$ for $(l \leq t \leq r_i)$ and $R_{l, r_i}(x_{r+1}) \leq R_{l, (r_i+1)}(x_{r+1})$. In other words, we claim that every element in sequence $Z'$ is greater than or equal to some element in $Z$.

Proof. Consider some node $x_t (l \leq t \leq r_i)$. From Eq. 10 we know that

$$R_{l, r_i}(x_t) = \max_{s \in \mathcal{S}} R_{l, r_i}(s, x_t)$$

27
where \( R_{i,r_i}(s, x_t) \) is the regret with ADP \( P_i \) for sink \( x_t \) under scenario \( s \). Let \( s^* \) be the worst-case scenario which maximizes the term \( R_{i,r_i}(s, x_t) \). Since \( s^* \) is a subpath of \( P_i' \), the evacuation time for sink \( x_t \) under scenario \( s^* \) in \( P_i \) is lesser than or equal to that in \( P_i' \),

\[
\Theta^1(P_i, x_t, s^*) \leq \Theta^1(P_i', x_t, s^*)
\]

\[
\Theta^1(P_i, x_t, s^*) - \Theta^{k}_{\text{opt}}(P, s^*) \leq \Theta^1(P_i', x_t, s^*) - \Theta^{k}_{\text{opt}}(P, s^*)
\]

\[
R_{i,r_i}(x_t, s^*) \leq R_{i,(r_i+1)}(x_t, s^*)
\]

Since, \( s^* \) is the maximizing term for \( R_{i,r_i}(x_t) \),

\[
R_{i,r_i}(x_t) \leq R_{i,(r_i+1)}(x_t, s^*)
\]

\[
\leq \max_{s \in S} R_{i,(r_i+1)}(x_t, s)
\]

\[
= R_{i,(r_i+1)}(x_t)
\]  \( \text{(42)} \)

Thus, for \( x_t(l_i \leq t \leq r_i) \), \( R_{i,r_i}(x_t) \leq R_{i,(r_i+1)}(x_t) \). Consider \( s^* \) to be the worst-case scenario which maximizes the regret \( R_{i,r_i}(s, x_{r_i}) \). Under scenario \( s^* \), for sink \( x_{r_i} \in P_i \), the evacuation time in \( P_i \) is equal to the left evacuation time as there are no people to evacuate to the right. Similarly, for sink \( x_{r_i+1} \in P_i' \), the evacuation time in \( P_i' \) equals the left evacuation time. But since \( x_{r_i+1} \) lies to the right of \( x_{r_i} \), the left evacuation time for \( x_{r_i+1} \) will be greater than or equal to the left evacuation time for \( x_{r_i} \), i.e.,

\[
\Theta^1(P_i, x_{r_i}, s^*) \leq \Theta^1(P_i', x_{r_i+1}, s^*)
\]

\[
\Theta^1(P_i, x_{r_i}, s^*) - \Theta^{k}_{\text{opt}}(P, s^*) \leq \Theta^1(P_i', x_{r_i+1}, s^*) - \Theta^{k}_{\text{opt}}(P, s^*)
\]

\[
R_{i,r_i}(x_{r_i}, s^*) \leq R_{i,(r_i+1)}(x_{r_i+1}, s^*)
\]

Since, \( s^* \) is the maximizing term for \( R_{i,r_i}(x_{r_i}) \),

\[
R_{i,r_i}(x_{r_i}) \leq R_{i,(r_i+1)}(x_{r_i+1}, s^*)
\]

\[
\leq \max_{s \in S} R_{i,(r_i+1)}(x_{r_i+1}, s)
\]

\[
= R_{i,(r_i+1)}(x_{r_i+1})
\]  \( \text{(43)} \)

From Eqs. 42 and 43 we can see that all the terms in sequence \( Z' \) is greater than or equal to some term in sequence \( Z \).

\( \square \)

**Corollary 1.** The minimum of the terms in sequence \( Z = R_{i,r_i} \) is lesser than or equal to the minimum of the terms in \( Z' = R_{i,(r_i+1)} \) (Follows from the previous claim).

Thus, \( R_{i,r_i} \leq R_{i,(r_i+1)} \) and by a symmetrical argument, \( R_{i,r_i} \leq R_{l_i-1,r_i} \).

\( \square \)
E.2 Proof of Lemma 4

Let \( R = \max_{1 \leq i \leq q} \{ R_{l,r} \} \) and \( R' = \max_{1 \leq i \leq q} \{ R'_{l,r} \} \). We have to prove that \( R \leq R' \).

Assume the contrary, i.e., \( R > R' \), i.e.,

\[
\max_{1 \leq i \leq q} \{ R_{l,r} \} > \max_{1 \leq i \leq q} \{ R'_{l,r} \}
\]

Now, let us consider the \( q \)-partition \( \hat{P}_q = \{ P_1, P_2, ..., P_q \} \). Now, consider the \( q \)-partition of \( \hat{P}_q' \) but only upt to \( x_{r_q} \), i.e., restricting the \( q^{th} \) part to end at \( x_{r_q} \). Let us call this \( \hat{P}_q'' \). In the \( q \)-partition \( \hat{P}_q'' \), the \( q^{th} \) part is smaller than the \( q^{th} \) part in \( \hat{P}_q' \) (by 1 node). Therefore, by Lemma 3, the minimax-regret for the \( q^{th} \) part being the ADP in \( \hat{P}_q'' \) is lesser than or equal to the minimax-regret for the \( q^{th} \) part as ADP in \( \hat{P}_q' \). All the other parts in \( \hat{P}_q'' \) have the same ADP minimax-regret as the parts in \( \hat{P}_q' \). Therefore, if \( R'' \) denotes the minimax-regret for the \( q \)-partition \( \hat{P}_q'' \), then \( R'' \leq R' \). This implies that \( R'' < R \), which contradicts our assumption that \( R \) was the minimax-regret considering the first \( q \) parts as ADPs with right end of the \( q^{th} \) part fixed at \( x_{r_q} \).

Therefore, \( R \leq R' \).

F Proofs for Minimax-Regret Binary-Search Based Algorithm

F.1 Calculation of \( R_{l,r} \) for all \( P_i \) in \( O(kn^2 \log n) \) time

From Property 1, we know that there are \( O(n) \) candidate worst-case scenarios given the \( k \)-partition \( \hat{P} \). Let \( \mathcal{S}_{\hat{P}} \) be the set of these \( O(n) \) scenarios. In order to find the minimax-regret given the \( k \)-partition \( \hat{P} \), we can use the following simple procedure:

1. For every scenario \( s^*_b \in \mathcal{S}_{\hat{P}} \) (\( O(n) \) scenarios):
   - Find the optimal \( k \)-sink evacuation solution for \( s^*_b \), \( O(n.k \log n) \) time) and store in \( \Theta^{k}_{\text{opt}}(P, s^*_b) \).

2. For every part \( P_i (0 \leq i \leq k) \):
   (a) For every candidate scenario \( s^*_b \in \mathcal{S}_{\hat{P}} \) (\( O(n) \) scenarios by Property 1):
      - Find \( R_{l,r_i}(s^*_b, x_j) \forall x_j \in P_i \) (in \( O(n) \) time, see Appendix F.3).
   3. For every possible sink \( x_j (0 \leq j \leq n) \) (\( O(n) \) sinks):
      - Find \( R_{l,r_i}(x_j) \) by finding max\( s^*_b \in \mathcal{S}_{\hat{P}} \) \( R_{l,r_i}(s^*_b, x_j) \) (\( O(n) \) time).
   4. For every part \( P_i (0 \leq i \leq k) \):
      - Find \( R_{l,r_i} \) by finding min\( l \leq j \leq r, R_{l,r_i}(x_j) \) (\( O(n) \) time for all parts combined)

5. The minimax-regret is the maximum of all \( R_{l,r_i} \) values. \( O(n) \) time

As we can see, this simple procedure to find the minimax-regret \( R_{l,r_i} \) for all \( P_i \) given the \( k \)-partition \( \hat{P} \) takes \( O(kn^2 \log n) \) time.
F.2 Proof of Property 3

If the parts \( \{P_{i+1}, P_{i+2}, \ldots, P_k\} \) are fixed, then consider the left end of the \( i^{th} \) part to be at \( x_l^i \). If the minimax-regret for the entire path is \( R \), we can write it as:

\[
R = \max_{1 \leq j \leq k} \{R_l r_j\} = \max \left( \max_{1 \leq j \leq i-1} R_l r_j, \max_{i \leq j \leq k} R_l r_j \right) \tag{44}
\]

Now if we move the left end of the \( i^{th} \) part to \( x_l^{i+1} \), then by Lemma 3, we know that \( R_{(l+1)r_i} \leq R_{l r_i} \). The values of \( R_l r_j \) for \( j = i+1, \ldots, k \) do not change as those parts do not change. This means the term \( \max_{i \leq j \leq k} R_l r_j \) in Eq. (44) cannot increase. Also, by Lemma 4, we know that the minimax-regret considering only the first \( i-1 \) parts as ADPs cannot decrease. This means that in Eq. (44) the term \( \max_{i \leq j \leq i-1} R_l r_j \) cannot decrease. Therefore, \( R \) is unimodal as a function of the left end of the \( i^{th} \) part if the parts \( \{P_{i+1}, \ldots, P_k\} \) are fixed.

\[\Box\]

F.3 Algorithm to find \( R_{l r_i}(s_B^*, x_j) \forall x_j \in P_i \) in \( O(n) \) time

Given a worst-case scenario \( s_B^* \in S_B^* \) (or any arbitrary scenario in fact) and an Assumed Dominant Part (ADP) \( P_i \in \hat{P} \) with left end (resp. right end) as \( x_l \) (resp. \( x_r \)), we first give a simple procedure to calculate \( \Theta_{L}(P_i, x_j, s_B^*) \) for all possible sinks \( x_j \in P_i \) under scenario \( s_B^* \).

We first prove the following lemma which will help us in proposing an efficient procedure:

**Lemma 6.** Consider a subpath (part) \( P_i \in \hat{P} \) and a sink \( x_j \in P_i \). Let us consider the left evacuation time which is the maximum of the evacuation functions of the vertices to the left of \( x_j \) (from Eq. [3]). Let \( m \) (resp. \( x_m \)) be the index (resp. vertex) which maximizes this function. If we move the sink from \( x_j \) to \( x_{j+1} \), then the maximizing index (resp. vertex) for the left evacuation to \( x_{j+1} \) would be either \( m \) (resp. \( x_m \)) or \( j \) (resp. \( x_j \)). The same statement holds symmetrically for the right evacuation time.

**Proof.** From Eq. [3] the left evacuation time for sink \( x_j \in P_i \) is:

\[
\Theta_{L}(P_i, x_j, s_B^*) = \max_{l \leq z \leq r} \left\{ (x_j - x_l)\tau + \sum_{l \leq z \leq t} w_z(s_B^*) \bigg| x_j > x_l \right\}
\]

and the maximizing term is \( t = m \). Now, when the sink is moved to \( x_{j+1} \), notice that the only change in the evacuation function of a node is the addition of the length \( |x_{j+1} - x_j| \). Therefore, the evacuation function among the nodes \( \{x_l, \ldots, x_{j-1}\} \) is still maximum for \( x_m \). Also, the evacuation function of one new node \( x_j \) (the previous sink) is added as a candidate for the maximum term.
Therefore, the new maximum can be calculated by one comparison of the evacuation function between the old maximum (for $x_m$) and newly added term (for $x_j$).

The following procedure will find $\Theta^1(P_i, x_j, s_B^*)\forall j(l_i \leq j \leq r_i)$ given scenario $s_B^*$:

1. For $x_j \in P_i$ (Iterating from $j = l_i$ to $r_i$):
   - Find left evacuation time for $x_j$ in $O(1)$ time (Lemma 6)
2. For $x_j \in P_i$ (Iterating back from $j = r_i$ to $l_i$):
   - Find right evacuation time for $x_j$ in $O(1)$ time (Lemma 6)
3. For $x_j \in P_i$:
   - Use the max of left and right evacuation times to find the evacuation time for $x_j$ under scenario $s_B^*$.

From this procedure, we can find out $\Theta^1(P_i, x_j, s_B^*)$, the evacuation time in $P_i$ under sink $x_j\forall j(l_i \leq j \leq r_i)$ in $O(n)$ time. Since we already know $\Theta^k_{opt}(P, s_B^*)$ (from the previous step of the procedure in Sec. 5), the optimal $k$-sink evacuation time for $s_B^*$, $R_{l_i r_i}(s_B^*, x_j) = \Theta^1(P_i, x_j, s_B^*) - \Theta^k_{opt}(P, s_B^*)$ can be determined in $O(n)$ time.

G An $O(kn)$ DP Implementation for the Minimax-Regret $k$-Sink Location Problem

G.1 DP Algorithm for Minimax-Regret $k$-Sink Location

A naive DP table filling procedure of the recurrence in Eq. 14 will take $O(kn^2)$ time. Also, the minimax-regret $k$-partition and sinks can be reconstructed by storing the optimizing values during the DP without adding any extra time complexity. We observe two crucial properties (very similar to the Properties 4 and 5) which helps bring down the running time.

Property 6. In the recurrence given by Eq. 14 keeping $q$(number of parts) fixed, if we increment $i$, then the minimizing $j$ value cannot decrease.

Proof. See Appendix G.2

Property 7. In the recurrence given by Eq. 14 keeping $i$ and $q$ fixed, max $(M(q-1,j-1), R_{ji})$ is unimodal with a unique minimum value as a function of $j$.

Proof. See Appendix G.3

Therefore, because of the two properties stated above, we observe that Remark 1 also holds true for this recurrence thereby allowing us to update $j$ in an amortized $O(1)$ time. The new recurrence is:

$$M(q, i) = \min_{j' \leq j \leq i} \max (M(q-1, j-1), R_{ji})$$

where $j'$ is the optimum value of $j$ for $M(q, i - 1)$.

The DP procedure is illustrated below:
1. Fill in \( M(1, i) = R_{0i} (0 \leq i \leq n) \). \((O(n) \text{ time})\)

2. For \( q \leftarrow 2 \) to \( k \):
   - \( j \leftarrow 0 \)
     - For \( i \leftarrow 0 \) to \( n \):
       - Do \( j \leftarrow j + 1 \) till minimum of \( \max(M(q - 1, j - 1), R_{ji}) \) is found
         which is equal to \( M(q, i) \).
       - Store \( M(q, i) \) in the DP table.

For a given value of \( q \) (number of parts), \( i \) and \( j \) are incremented atmost \( n \) times each and each increment is handled in \( O(1) \) time. Therefore, the running time is \( O(kn) \).

\[ \text{G.2 Proof of Property 6} \]

The proof for this property is very similar to the proof for Property 4 because of the similar nature the optimal evacuation time recurrence and the minimax-regret recurrence.

Equation 14 is given by:

\[
M(q, i) = \min_{0 \leq j \leq i} \max(M(q - 1, j - 1), R_{ji})
\]

where \( R_{ji} \) is the minimax-regret with ADP(Assumed Dominant Part) as the part with left end \( x_j \) and right end \( x_i \).

Assume \( j' \) is the minimizing \( j \) value for \( M(q, i - 1) \) and \( j'' \) is the minimizing \( j \) value for \( M(q, i) \). We need to prove that \( j'' \geq j' \).

Assume the contrary, i.e., \( j'' < j' \). We now have two cases to deal with:

**Case 1:** \( R_{j'(i-1)} \geq M(q - 1, j' - 1) \)

In this case, the minimax-regret calculated in the subpath from \( x_0 \) to \( x_{i-1} \) for \( q \) parts will be \( M(q, i - 1) = R_{j'(i-1)} \). Now,

\[
M(q, i) = \max(M(q - 1, j''), R_{j''i})
\]

**Remark 4.** Consider the minimax-regret considering only the first \( q \) parts as ADPs. This minimax-regret calculated when these first \( q \) parts are restricted to a path is greater than or equal to the minimax-regret when the first \( q \) parts are restricted to any of its subpaths.

**Proof.** This follows from Lemma 4 where it is proved when a path is extended to the right by a node. By symmetry, it is also true when the path is extended to the left. \( \square \)

Since \( j'' < j' \),

\[
M(q - 1, j'' - 1) \leq M(q - 1, j' - 1) \quad \text{(from Remark 4)}
\]

\[
\leq R_{j'(i-1)} \quad \text{(from case assumption)}
\]

\[
\leq R_{j'i} \quad \text{(from Remark 4)}
\]

\[
\leq R_{j''i} \quad \text{(from Remark 4)}
\]

(47)
Therefore from Eq. 47, \( M(q, i) = R_{j''i} \). Since \( j'' \) is the largest minimizing \( j \) value for \( M(q, i) \) and \( j'' > j' \),
\[
R_{j''i} < \max (M(q - 1, j' - 1), R_{j'i}) \quad (48)
\]
\[
= R_{j'i} \quad (49)
\]
But from Lemma 3, we know that \( R_{j''i} \geq R_{j'i} \). Eq. 48 cannot be true. We arrive at a contradiction. Therefore, our assumption that \( j'' < j' \) is false for this case. So, \( j'' \geq j' \).

**Case 2:** \( R_{j'(i-1)} < M(q - 1, j' - 1) \)
In this case, \( M(q, i - 1) = M(q - 1, j' - 1) \). The minimizing \( j \) value for \( M(q, i) \) is \( j'' \),
\[
M(q, i) = \max (M(q - 1, j'' - 1), R_{j''i})
\]
Since \( j'' < j' \),
\[
\max (M(q - 1, j'' - 1), R_{j''i}) < \max (M(q - 1, j' - 1), R_{j'i}) \quad (50)
\]
By the case assumption and the fact that \( j' \) is the minimizing \( j \) value for \( M(q, i - 1) \),
\[
R_{j'(i-1)} < M(q - 1, j' - 1) \quad \text{(from case assumption)}
\]
\[
\leq \max (M(q - 1, j'' - 1), R_{j''(i-1)}) \quad \text{(from Remark 4)}
\]
\[
\leq \max (M(q - 1, j'' - 1), R_{j''i}) \quad (51)
\]
Therefore,
\[
M(q - 1, j' - 1) \leq \max (M(q - 1, j'' - 1), R_{j''i}) \quad \text{(from Eq. 51)} \quad (52)
\]
\[
R_{j'i} \leq R_{j''i} \quad \text{(from Remark 4)} \quad (53)
\]
It is easy to see that Eqs. 50, 52 and 53 contradict each other. Therefore, our assumption that \( j'' < j' \) is false for this case. So, \( j'' \geq j' \).

**G.3 Proof of Property 7**

The minimax-regret is given by the recurrence in Eq. 14:
\[
M(q, i) = \min_{0 \leq j \leq i} \{ \max (M(q - 1, j - 1), R_{ji}) \}
\]
Keeping \( q \) and \( i \) fixed, if we increment \( j \), then by Lemma 3 \( R_{ji} \) cannot increase and by Lemma 4 \( M(q - 1, j - 1) \) cannot decrease. This implies that \( \max (M(q - 1, j - 1), R_{ji}) \) is unimodal as a function of \( j \). □

**H Reduction of \( R_{ji} \) Precomputation Time from \( O(n^5) \) to \( O(n^3) \)**

In Sect. 7.1, we had given an \( O(n^5) \) procedure for the precomputation of \( R_{ji} \)'s. Here, we will reduce the precomputation time to \( O(n^3) \).
H.1 A Lookup Table for Step 4 - $O(n^4)$

In Step 4 of the Naive approach (in Sect. [7.1], we take $O(n)$ time to compute the regret because of the calculation of the evacuation time on a subpath given a sink and a scenario. Instead, we can construct an $O(1)$ time lookup data structure in $O(n^3)$ time. A (subpath,sink,scenario) query to the structure will yield the 1-sink evacuation time in the subpath for the sink under the scenario in $O(1)$ time (For details see Appendix [11]). Now, finding the regret for a (subpath,sink,scenario) triplet can be done in $O(1)$ time. Thus, the time for precomputation of $R_{ji}$ comes down to $O(n^4)$.

H.2 Amortizaton on Step 2 - $O(n^3)$

In Step 2, for each subpath, we are checking $O(n)$ possible minimax-regret sink locations. Instead, if the following two lemmas were true, we would only need to check an amortized $O(1)$ sink locations.

**Lemma 7.** Consider an ADP (Assumed Dominant Part) with leftmost(resp. rightmost) node as $x_l$(resp. $x_r$). Let $x_t(l \leq t \leq r)$ be the minimax-regret sink which minimizes the max-regret. In the subpath from $x_l$ to $x_{r+1}$ (extending the right end of the part), there exists a minimax-regret sink at some $x_i$, where $i \geq t$.

*Proof. See Appendix [12]*

**Lemma 8.** Consider an ADP with leftmost(resp. rightmost) node as $x_l$(resp. $x_r$). The max-regret for a sink $x_i(l \leq i \leq r)$ is $R_{li}(x_i)$. $R_{li}(x_i)$ is unimodal with a unique minimum value as a function of the sink $x_i(l \leq i \leq r)$.

*Proof. See Appendix [13]*

By using Lemmas [7] and [8] we can see that by appending a node to the right of a subpath, the minimax-regret sink cannot move to the left and it is possible to locate the minimax-regret sink in the subpath by scanning linearly to right of the previous minimax-regret sink. Thus, the precomputation of $R_{ji}$’s can be rewritten to be completed in $O(n^3)$ time as follows:

1. Consider some $x_l$ as the left end of the subpath, move the right-end $x_r$ away from $x_l$. There are $O(n)$ possible right-ends to a left end $x_l$.
2. By Lemmas [7] and [8] the minimax-regret sink $x_1$ cannot move to the left as the right-end $x_r$ increases. Therefore over all possible right-ends $x_r$, $t$(index of current minimax-regret sink) is incremented only $O(n)$ times, i.e., only an amortized $O(1)$ sink locations need to be checked for a right-end $x_r$.
3. Each candidate minimax-regret sink has $O(n)$ possible worst-case scenarios $s^*_B$. The regret can be looked up in $O(1)$ time because of the precomputation in Sect. [H.1]. The max-regret for the sink over all possible worst-case scenarios $s^*_B$ can be calculated in $O(n)$ time.
4. There are $O(n)$ possible left ends $x_l$ to consider.
5. Therefore, the total running time in $O(n^3)$. 

34
Proofs for the precomputation of $R_{ji}$’s

I. Construction of Lookup Table for the 1-sink evacuation time for any (subpath,sink,scenario) triplet in $O(n^3)$ time

We are going to show a method by which we can construct a $O(1)$ time lookup structure for the evacuation time of any subpath with left end $x_l$ and right end $x_r$, for a sink $x_t(l \leq t \leq r)$ within the subpath under all candidate worst-case scenarios in $s^*_B \in S^*$. 

Now, according to the proof of Lemma given in Appendix, the worst-case scenario for a sink in a subpath is either left-dominant or right-dominant and the weights of the vertices transform from $w^{-i}$ to $w^{+i}$ at a node $x_m$ which maximizes the evacuation function for that sink (Refer Fig. 3). Therefore, a left-dominant or right-dominant scenario in a subpath from $x_l$ to $x_r$ with sink $x_t$ can be represented by a vertex $x_m$ which is the transition point of the weights from $w^{-i}$ to $w^{+i}$.

We are going to construct two tables:

- $L(x_l, x_m, x_t)$ - Stores the evacuation time in subpath from $x_l$ to $x_t$ under a left-dominant scenario with transition vertex $x_m(l \leq m < t)$.
- $R(x_t, x_m, x_r)$ - Stores the evacuation time in subpath from $x_t$ to $x_r$ under a right-dominant scenario with transition vertex $x_m(t < m \leq r)$.

Conceptually, tables $L$ and $R$ store the left and right evacuation time for a sink, subpath and scenario. The evacuation time in a subpath from $x_l$ to $x_r$ for a sink $x_t(l \leq t \leq r)$ under a candidate worst-case scenario which has transition vertex $x_m$ is:

- $\max(L(x_l, x_m, x_t), R(x_t, x_m, x_r))$, if $x_m < x_t$ (a left-dominant scenario), or
- $\max(L(x_l, x_t, x_t), R(x_t, x_m, x_r))$, if $x_m > x_t$ (a right-dominant scenario).

Therefore, any (subpath,sink,scenario) query for the evacuation time in a subpath for a sink under a scenario can be looked up in $O(1)$ time if we have the tables $L$ and $R$.

We now show how to calculate the table $R(x_t, x_m, x_r)$. Table $L(x_l, x_m, x_t)$’s construction is symmetric. For a given sink $x_t$ and the right end $x_r$, $x_m(t < m \leq r)$ has a possible $O(n)$ locations. The following procedure fills up the $R$ table:

1. For $t \gets 0$ to $n-1$: ($O(n)$)
   a. For $r \gets t+1$ to $n$: ($O(n)$)
      i. For $m \gets t+2$ to $r$: ($O(n)$)
         - $R(x_t, x_m, x_r) = R(x_t, x_m-1, x_r) + |x_m - x_{m-1}| - w_{m-1}$

A similar procedure can be used to fill up the table $L(x_l, x_m, x_t)$. The running time for construction of $L$ and $R$ is $O(n^3)$. 

35
I.2 Proof for Lemma 7

Let us first define a few notations we will be using in this proof:

- $P^O$ is the subpath with leftmost (resp. rightmost) node $x_l$ (resp. $x_r$).
- $P^N$ is the subpath with leftmost (resp. rightmost) node $x_l$ (resp. $x_{r+1}$).
- $x_i$ (resp. $x$) is the the minimax-regret sink in subpath $P^O$ (resp. $P^N$).
- For node $x_j$ in subpath $P^O$ (resp. $P^N$), $R^O_j$ (resp. $R^N_j$) is the minimax-regret if $x_j$ is chosen as the sink.
- $\Theta^O_L(s, x_j)$ (resp. $\Theta^N_L(s, x_j)$) is the left evacuation time in subpath $P^O$ (resp. $P^N$) if $x_j$ is chosen as the sink under scenario $s$. Similarly, $\Theta^O_R(s, x_j)$ and $\Theta^N_R(s, x_j)$ is defined.
- $\Theta^O(s, x_j)$ (resp. $\Theta^N(s, x_j)$) is the overall evacuation time in subpath $P^O$ (resp. $P^N$) if $x_j$ is chosen as the sink under scenario $s$.

Let us assume the contrary, i.e., there exists no minimax-regret sink $x_i \in P^N$ such that $i \geq t$, i.e., $i < t$. Therefore,

$$R^N_i < R^N_t$$

since the minimax-regret sink in $P^N$ is not located at $i = t$.

Since $x_t$ is the minimax-regret sink in $P^O$,

$$R^O_t \leq R^O_i$$

Now, since $P^N$ is $P^O$ appended by a node, the following equation holds:

$$R^O_i \leq R^N_i$$

Equations 55, 54 and 56 gives us,

$$R^O_t < R^N_i$$

Let $s^N_t$ be the worst-case scenario for $x_t$ in subpath $P^N$.

Claim. $\Theta^O_R(s^N_t, x_t) \geq \Theta^N_R(s^N_t, x_t)$, i.e., in subpath $P^N$, for sink $x_t$, the right evacuation time is greater than or equal to the left evacuation time under its worst-case scenario $s^N_t$.

Proof. Assume the contrary, i.e., $\Theta^O_R(s^N_t, x_t) < \Theta^N_R(s^N_t, x_t)$. Now,

$$R^N_t = \Theta^O_L(s^N_t, x_t) - \Theta^O_R(s^N_t, x_t) = \Theta^N_L(s^N_t, x_t) - \Theta^N_R(s^N_t, x_t) \leq R^O_i$$

This contradicts Equation 57. Our assumption is false and therefore, $\Theta^O_R(s^N_t, x_t) \geq \Theta^N_R(s^N_t, x_t)$.

Since $i < t$, by Claim 1.2,

$$\Theta^N(s^N_t, x_i) \leq \Theta^N(s^N_t, x_t)$$

$$\implies R^N_i \leq R^N_t$$

This contradicts Equation 54. Therefore our assumption that there exists no minimax-regret sink $x_i$ ($i \geq k$) is false. Thus there exists a minimax-regret sink $x_i$ in $P^N$ such that $i \geq t$. 

\[\square\]
\section*{I.3 Proof for Lemma 8}

Consider $P_{lr}$ to be the subpath from node $x_l$ to $x_r$. For any given scenario $s \in S$, the evacuation time in subpath $P_{lr}$ will be unimodal with unique minimum value as a function of the sink $x_i$, i.e., $\Theta^1(P_{lr}, x_i, s)$ is unimodal.

By definition (in Eq. 9),

$$R_{lr}(s, x_i) = \Theta^1(P_{lr}, x_i, s) - \Theta^k_{\text{opt}}(P, s).$$

Therefore, $R_{lr}(s, x_i)$ is also unimodal with a unique minimum value as a function of $x_i$. Also the max-regret can be defined as (by Eq. 10),

$$R_{lr}(x_i) = \max_{s \in S} R_{lr}(s, x_i)$$

$R_{lr}(x_i)$ is the maximum of $O(n)$ unimodal functions with unique minimum values. Therefore, $R_{lr}(x_i)$ is also unimodal function with a unique minimum value as a function of $x_i (l \leq i \leq r)$.

\qed