




# Minmax Regret $k$ -Sink Location on a Dynamic Path Network with Uniform Capacities

Guru Prakash Arumugam<sup>1</sup> · John Augustine<sup>2</sup> · Mordecai J. Golin<sup>3</sup>  · Prashanth Srikanthan<sup>4</sup>

Received: 20 September 2018 / Accepted: 23 May 2019 / Published online: 4 June 2019  
© Springer Science+Business Media, LLC, part of Springer Nature 2019

## Abstract

In *Dynamic flow networks* an edge's capacity is the amount of flow (items) that can enter it in unit time. All flow must be moved to sinks and *congestion* occurs if flow has to wait at a vertex for other flow to leave. In the *uniform capacity* variant all edges have the same capacity. The minmax  $k$ -sink location problem is to place  $k$  sinks minimizing the maximum time before all items initially on vertices can be evacuated to a sink. The *minmax regret* version introduces uncertainty into the input; the amount at each source is now only specified to within a range. The problem is to find a universal solution (placement of  $k$  sinks) whose regret (difference from optimal for a given input) is minimized over all inputs consistent with the given range restrictions. The previous best algorithms for the minmax regret version of the  $k$ -sink location problem on a path with uniform capacities ran in  $O(n)$  time for  $k = 1$ ,  $O(n \log^4 n)$  time for  $k = 2$  and  $\Omega(n^{k+1})$  for  $k > 2$ . A major bottleneck to improving those solutions was that minmax regret seemed an inherently *global* property. This paper derives new combinatorial insights that allow decomposition into *local* problems. This permits designing two new algorithms. The first runs in  $O(n^3 \log n)$  time independent of  $k$  and the second in  $O(nk^2 \log^{k+1} n)$  time. These improve all previous algorithms for  $k > 1$  and, for  $k > 2$ , are the first polynomial time algorithms known.

**Keywords** Dynamic flow networks · Sink location problems · Evacuation · Minmax regret

---

Guru Prakash Arumugam: Work done while at the Indian Institute of Technology Madras (IIT Madras) and when interning at Hong Kong UST. John Augustine: Work supported in part by an Extra Mural Research Grant (EMR/2016/003016) and a MATRICS Project (MTR/2018/001198) both funded by SERB, DST, India. Mordecai J. Golin: Work partially supported by RGC CERG 16208415. Prashanth Srikanthan: Work done while at IIT Madras and when interning at Hong Kong UST.

---

✉ Mordecai J. Golin  
golin@cs.ust.hk

Extended author information available on the last page of the article

## 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 between neighboring vertices. Each edge  $e$  has a given capacity  $c_e$  which limits the rate of the flow of supplies *entering* that edge in unit time. If there is some  $c > 0$  such that  $\forall e, c_e \equiv c$ , the network has *uniform capacity*. Each  $e$  also has associated with it a time required to traverse  $e$ . Note that as supplies move around the graph, *congestion* can occur, as supplies back up at a vertex.

Dynamic flow networks were introduced by Ford and Fulkerson in [18] and have since been extensively researched. One well studied problem on such networks is transshipment, e.g., [23], in which the graph has several sources and sinks, with supplies on the sources and each sink having a specified demand. The problem is to find the quickest time required to satisfy all of the demands. Hoppe and Tardos [23] provide a polynomial time algorithm for this problem.

Dynamic Flow also models *evacuation problems* [21]. Vertices model buildings and the items on vertices represent evacuees in buildings. Since our work also fits this evacuation analogy, we will refer to items on vertices as evacuees, and edges are roads connecting buildings. The sinks play the role of exits to safety from the roads. The problem is to find a routing strategy (evacuation plan) that evacuates everyone to the sinks in minimum time. In these problems the evacuation plan is vertex based. That is, all supplies from a vertex evacuate out through a single edge given by the plan (corresponding to a sign at that vertex stating “this way out”). Such a flow, in which all flow through a vertex leaves through the same edge, is known as *confluent*. The basic minmax optimization problem is to determine a plan that minimizes the total time needed to evacuate all the evacuees. Confluent flows are known to be difficult on general graphs; solving, or even approximating to a constant factor, the quickest confluent flow problem with even one sink is NP-Hard [19,24]. Research on finding *exact* quickest confluent dynamic flows is therefore restricted to special graphs, such as trees and paths.

In some versions of the problem the sinks are known in advance. In others, such as the one addressed in this paper, locating the sinks that minimize the evacuation time is part of the problem; only the *number*  $k$  of allowed sinks is given as input. Mamada [28] gives an  $O(n \log^2 n)$  algorithm for solving the 1-sink problem on a dynamic tree network. Higashikawa et al. [21] improve this down to  $O(n \log n)$  for uniform capacities. For  $k$  sinks this can be solved in  $O(nk^2 \log^5 n)$  time for general capacities and  $O(nk^2 \log^4 n)$  for uniform capacities [15]. Higashikawa et al. [22] show how to solve the  $k$ -sink problem on a uniform capacity dynamic path network in  $O(kn)$  time which was later reduced down to  $O(\min(n \log n, n + k^2 \log^2 n))$  by [7]. We note that all the results above refer to the *minmax* problem of minimizing the maximum evacuation time. There is also a corresponding *minsum* problem of minimizing the total sum of all of the evacuation times of all of the flow. This can be solved on a dynamic path network [5] in time  $O(kn^2 \log^2 n)$  for general capacities and  $O(kn \log^3 n)$  for uniform capacities.

In practice, the exact number of evacuees located at a vertex is unknown at the time the plan is drawn up and *robust optimization* is needed. One approach to robust

optimization is probabilistic, with the number of evacuees at a vertex being drawn from known distributions. In another, the one used in this paper, all that is known is the (interval) range in which that number may fall. One way to deal with this type of uncertainty is to find a plan that minimizes the *regret*, e.g. the maximum discrepancy between the evacuation time for the given plan on a fixed input and the plan that would give the minimum evacuation time for that particular input. This is known as the *minmax regret* problem. Minmax regret optimization has been extensively studied for  $k$ -median [10,13,38] and  $k$ -center [3,11,32,38] ([12] is a recent example) along with many other optimization problems [17,33,37]. Aissi et al. [1], Averbakh and Lebedev [4], Candia-Véjar et al. [14] and Kouvelis and Gang [25] provide an introduction to the literature. Since most of these problems are NP-Hard to solve exactly on general graphs, the vast majority of the literature is again concerned with algorithms on special graphs, in particular paths and trees.

Recently there has been a series of new results for  $k$ -sink evacuation on special structure dynamic graphs. The 1-sink minmax regret problem on a uniform capacity path was originally proposed by [16] who gave an  $O(n \log^2 n)$  algorithm. This was reduced down to  $O(n \log n)$  by [20,34,35] and then to  $O(n)$  by [9]. Xu and Li solve the 1-sink minmax regret problem on a uniform capacity cycle in  $O(n^3 \log n)$  time [36]. Higashikawa et al. [21] provides an  $O(n \log^2 n)$  algorithm for the 1-sink minmax regret problem on a uniform capacity tree; this was reduced to  $O(n \log n)$  by [9].

Returning to the minmax regret problem uniform capacity *path* case; for  $k = 2$ , [27] gave an  $O(n^3 \log n)$  algorithm, later reduced to  $O(n \log^4 n)$  by [9].

For  $k > 2$  the only previous algorithm known was  $O(n^{1+k}(\log n)^{1+\log k})$  [30]. The exponential growth with  $k$  for that algorithm was an inherent property of the structural technique used.

We also note two other related sink regret problems. The first is the regret version of the  $k = 1$  minsum problem on a path with uniform capacities. This was originally solved in  $O(n^3)$  time by [5] and then reduced down to  $O(n^2 \log^2 n)$  time by [8]. The second is the original minmax regret problem with  $k = 1$  sink on a *general* graph with uniform capacities and confluent flows but with the problem restricted so that every vertex evacuates all of its flow along the *shortest* path to the sink. This can be solved [26] in  $O(m^2 n^3)$  time where  $m$  is the number of edges in the graph.

*Organization* This paper develops new structural properties of the minmax regret solution for paths, leading to better algorithms. Section 3 proves that, independent of  $k$ , there are only  $O(n^2)$  worst case scenarios that need to be checked. This independence permits overcoming the exponential dependence upon  $k$  in [30]. Section 4 develops properties of worst-case solutions that could lead to efficient algorithms. Section 5 develops further such properties. Section 6 develops two such algorithms, both of which improve on the previous ones in all  $k > 1$  cases. The first, better for small  $k$ , runs in  $O(nk^2(\log n)^{k+1})$ , which will be the best algorithm for any fixed  $k > 1$ . In particular for  $k = 2$ , it runs in  $O(n \log^3 n)$  time improving upon the  $O(n \log^4 n)$  algorithm of [9]. The second, which is better for large  $k$  (where  $k$  is dependent upon  $n$ ) runs in  $O(n^3 \log n)$  time with no dependence upon  $k$ . Section 7 discusses possible extensions. We emphasize that this paper assumes that all edges have the same capacity. This follows all prior work in this area e.g., on the path [9,16,20,34,35], trees [9,21] and

the cycle [36], which make the same uniform capacity assumption. Section 7 briefly discusses the structural reason *why* this and all previous work have not attempted to address the case in which different edges can have different capacities.

Finally, we note that our algorithms use as a subroutines procedures for efficiently calculating evacuation times. These procedures are straightforward modifications of well-developed techniques from [7,9,16] and contain very few new ideas. To keep this paper self contained, their proofs are provided in the “Appendix” in Sect. 8.

## 2 Preliminary Definitions

### 2.1 Model Definition

Let  $P = (V, E)$  be a path of  $(n + 1)$  vertices  $V = \{x_0, x_1, \dots, x_n\}$  along a line:  $x_0 < x_1 < \dots < x_n$ .  $E = \{e_1, e_2, \dots, e_n\}$  with  $e_i = (x_{i-1}, x_i)$  are the  $n$  edges connecting the  $x_i$ . For intuition we will sometimes refer to the vertices as *buildings* and the edges as *roads* connecting the buildings. Flow will correspond to evacuees.

We use  $d(x_i, x_j) = |x_i - x_j|$  to denote the distance between  $x_i$  and  $x_j$  and  $\tau$  to denote its *pace or inverse-speed*. Thus, travelling from  $x_i$  to  $x_j$  requires  $\tau d(x_i, x_j)$  time. All edges have fixed *capacity*  $c > 0$ ; this is the number of items that can enter an edge in unit time. Intuitively, this can be visualized as the road’s *width* (number of lanes), i.e., the number of items that can travel in parallel along the road.

Each vertex  $x_i$  also has a fixed but unknown weight  $w_i \geq 0$  denoting the number of people (evacuees) initially located at  $x_i$ . The input is a *range*  $[w_i^-, w_i^+]$  within which  $w_i$  must lie.

Let  $\mathcal{S}$  denote the Cartesian product of all weight intervals for  $0 \leq i \leq n$ :

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

A *scenario*  $s \in \mathcal{S}$  is an assignment of weights to all vertices. The *weight of a vertex*  $x_i$  *under scenario*  $s$  is denoted by  $w_i(s)$ .  $\mathcal{S}$  is the set of all scenarios consistent with the input restrictions.

Our arguments will often use “arg max”. In the case of *ties*, “arg max” returns the smallest index that achieves the maximum, i.e.,

$$\arg \max_{i \in \mathcal{I}} f(i) := \min \left\{ j \in \mathcal{I} : f(j) = \max_{i \in \mathcal{I}} f(i) \right\}.$$

### 2.2 Evacuation Time

Let  $P = (V, E)$  be a path with uniform edge capacity  $c$ , pace  $\tau$  and specified scenario  $s \in \mathcal{S}$ .

### 2.2.1 Left and Right Evacuation

Consider  $W$  units of flow starting at vertex  $x_i$  moving right. They require  $W/c + \tau(x_{i+1} - x_i)$  time until the last flow arrives at  $x_{i+1}$ ;  $W/c$  time before the last flow leaves  $x_i$  and  $\tau(x_{i+1} - x_i)$  further time until it arrives at  $x_{i+1}$ . If this flow is continuing on to  $x_{i+2}$ , it might or might not have to wait at  $x_{i+1}$  before entering edge  $[x_{i+1}, x_{i+2}]$ , depending upon whether other evacuees remain present on  $x_{i+1}$  at the time the first flow from  $x_i$  arrives there. It is these *congestion* effects that complicate the analysis.

Consider a single sink  $x \in P$  ( $x$  is not restricted to be a vertex in  $V$ ).

See Fig. 1. Define  $\Theta_L(P, x, s)$  (resp.  $\Theta_R(P, x, s)$ ) to be the time taken to evacuate everything to the left (right) of sink  $x$  to  $x$  under scenario  $s$ . The left (right) evacuation times have been derived by many authors, e.g., [9] to be

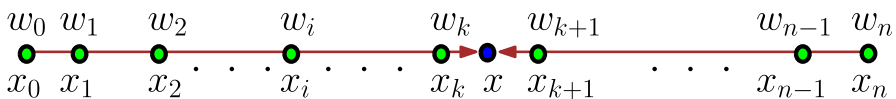
$$\Theta_L(P, x, s) = \max_{x_i < x : W_{0,i}^s > 0} \left\{ (x - x_i)\tau + \frac{W_{0,i}^s}{c} \right\} \tag{1}$$

$$\Theta_R(P, x, s) = \max_{x_i > x : W_{i,n}^s > 0} \left\{ (x_i - x)\tau + \frac{W_{i,n}^s}{c} \right\} \tag{2}$$

where for  $a \leq b$ ,  $W_{a,b}^s = \sum_{j=a}^b w_j(s)$ .

*Note* Intuitively,  $\Theta_L(P, x, s)$  is the time at which the last evacuee from  $x_0$  arrives at  $x$ . Let  $x_i < x$  be any vertex. The time required for the first evacuee leaving  $x_i$  to travel to  $x$ , without congestion, is  $T_1(i) = \tau(x - x_i)$ . The minimum time required for all evacuees in  $[x_0, x_i]$  to pass through  $x_i$  is  $T_2(i) = \frac{1}{c} W_{0,i}^s$ . Thus  $\Theta_L(P, x, s) \leq T_1(i) + T_2(i)$ . This is valid for every  $i$  so the right side of Eq. 1 is a lower bound for  $\Theta_L(P, x, s)$ . One can show that if  $x_{i^*}$  is the rightmost vertex at which the first evacuee from  $x_0$  experiences congestion by having to wait, then  $\Theta_L(P, x, s) = T_1(i^*) + T_2(i^*)$ , proving (1). The derivation of  $\Theta_R(P, x, s)$  is similar.

As stated, our setup assumes that sinks may be placed anywhere on the path and not just on a vertex. We note that our results will continue to hold even if sinks are restricted to be on vertices, i.e., our algorithms will still find the minmax regret  $k$ -sink location problem in the same amount of time under those new placement restrictions. Section 7 briefly discusses why this is true.



**Fig. 1** Illustration of left and right evacuation. Scenario  $s$  assigns  $w_j = w_j(s)$  initial units of flow to vertex  $x_j$ . All vertices are evacuated to a sink at  $x$ . The time  $\Theta_L(P, x, s)$  to evacuate everything to the left of  $x$  to  $x$  and the time  $\Theta_R(P, x, s)$  to evacuate everything to the right of  $x$  to  $x$  are given by Eqs. 1 and 2. Total evacuation time is the maximum of the two

### 2.2.2 1-Sink Evacuation

The *evacuation time to sink*  $x$  is the maximum of the left and right evacuation times: (Fig. 1)

$$\Theta^1(P, x, s) := \max\{\Theta_L(P, x, s), \Theta_R(P, x, s)\}. \tag{3}$$

### 2.2.3 $k$ -Sink Evacuation

**Definition 1** (Fig. 2) A separation of path  $P$  into  $k$  subpaths is denoted by  $\hat{P} = \{P_1, P_2, \dots, P_k\}$ ,  $P_i = [x_{l_i}, x_{r_i}]$  where  $l_1 = 0$ ,  $r_k = n$  and  $l_{i+1} = r_i + 1$ . Each  $P_i \in \hat{P}$  will be called a *partition*.  $\hat{Y} = \{y_1, y_2, \dots, y_k\}$  denotes a set of sinks such that  $y_i \in P_i$ . Given  $\hat{P}$  and sinks  $\hat{Y}$ , vertices in partition  $P_i$  are restricted to evacuate only to sink  $y_i$

Every vertex belongs to only one subpath. This implies that all evacuees passing through the same vertex must evacuate to the same sink, i.e., a *confluent* routing. The pair  $\{\hat{P}, \hat{Y}\}$  defines an *evacuation protocol* for the path  $P$ .

The  *$k$ -sink evacuation time with evacuation protocol  $\{\hat{P}, \hat{Y}\}$  under scenario  $s$*  is the maximum of the evacuation times within the individual partitions, i.e.,

$$\Theta^k(P, \{\hat{P}, \hat{Y}\}, s) := \max_{1 \leq i \leq k} \Theta^1(P_i, y_i, s) \tag{4}$$

**Definition 2** Given  $\hat{P}, \hat{Y}$ , set

$$d := \arg \max_{1 \leq i \leq k} \Theta^1(P_i, y_i, s).$$

The partition  $P_d$  with maximum evacuation cost (breaking ties to the leftmost) is called the “*dominant partition for  $\{\hat{P}, \hat{Y}\}$  under scenario  $s$* ”. We use  $y_d$  to denote “*the sink associated with the dominant partition  $P_d$* ”.

### 2.3 The Optimal $k$ -Sink Location Problem

The *Optimal  $k$ -Sink Location Problem* is to find the minimum evacuation cost using  $k$  sinks, i.e.,

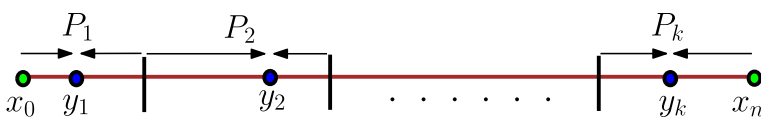


Fig. 2  $k$ -sink evacuation on Path  $P$  with partitions and sinks  $\{\hat{P}, \hat{Y}\}$

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

and an evacuation protocol  $\{\hat{P}^*, \hat{Y}^*\}$  that achieves this minimum, i.e.,

$$\Theta^k(P, \{\hat{P}^*, \hat{Y}^*\}, s) = \Theta_{\text{opt}}^k(P, s).$$

### 2.4 Regret

For fixed  $\{\hat{P}, \hat{Y}\}$  and  $s \in \mathcal{S}$ , the *regret* is defined as the difference between  $\Theta^k(P, \{\hat{P}, \hat{Y}\}, s)$  and the optimal  $k$ -sink evacuation time for  $s$ , i.e.,

$$R(\{\hat{P}, \hat{Y}\}, s) := \Theta^k(P, \{\hat{P}, \hat{Y}\}, s) - \Theta_{\text{opt}}^k(P, s) \tag{5}$$

The following self-evident property will be needed later:

**Property 1** If  $P_d$  is the dominant partition for  $\{\hat{P}, \hat{Y}\}$  under scenario  $s$ , then

$$R(\{\hat{P}, \hat{Y}\}, s) = \Theta^1(P_d, y_d, s) - \Theta_{\text{opt}}^k(P, s) \geq 0.$$

**Definition 3** The maximum regret (called *max-regret*) achieved (over all scenarios) for a choice of  $\{\hat{P}, \hat{Y}\}$  is:

$$R_{\text{max}}(\{\hat{P}, \hat{Y}\}) := \max_{s \in \mathcal{S}} \{R(\{\hat{P}, \hat{Y}\}, s)\} \tag{6}$$

If  $R_{\text{max}}(\{\hat{P}, \hat{Y}\}) = R(\{\hat{P}, \hat{Y}\}, s^*)$  for some scenario  $s^* \in \mathcal{S}$ , then  $s^*$  is a *worst-case scenario* for  $\{\hat{P}, \hat{Y}\}$ .

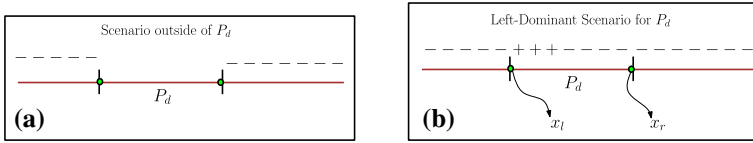
**Definition 4** The  $k$ -sink minmax regret value for  $P$  is

$$R_{\text{max}}^k(P) := \min_{\{\hat{P}, \hat{Y}\}} R_{\text{max}}(\{\hat{P}, \hat{Y}\}).$$

### 2.5 The Minmax Regret $k$ -Sink Location Problem

The input for the minmax regret  $k$ -Sink Location Problem is a dynamic path network with path  $P$ , vertex weight intervals  $[w_i^-, w_i^+]$ , edge capacity  $c$ , and pace  $\tau$ . The problem can be viewed as a 2-person Stackelberg game [31, 97–98] between the algorithm  $A$  and adversary  $B$ :

1. Algorithm  $A$  (leader): creates an evacuation protocol  $\{\hat{P}, \hat{Y}\}$  as defined in Sect. 2.2.3.
2. Adversary  $B$  (follower): chooses a regret-maximizing *worst-case scenario*  $s^* \in \mathcal{S}$  for  $\{\hat{P}, \hat{Y}\}$  i.e.,  $R(\{\hat{P}, \hat{Y}\}, s^*) = R_{\text{max}}(\{\hat{P}, \hat{Y}\})$ .



**Fig. 3** Illustrations of Lemma 1 and Definition 5. “+”s denote that  $w_i(s) = w_i^+$ . “-”s denote that  $w_i(s) = w_i^-$

$A$ 's objective is to find  $R_{\max}^k(P)$  and an evacuation protocol  $\{\hat{P}^*, \hat{Y}^*\}$  that minimizes this max-regret, i.e.,

$$R_{\max}^k(P) = R_{\max}(\{\hat{P}^*, \hat{Y}^*\}).$$

### 3 Worst Case Scenarios and Their Properties

A priori, every  $\{\hat{P}, \hat{Y}\}$  might have a different associated worst case scenario. This section describes an  $O(n^2)$  size set of scenarios  $\mathcal{S}^*$  that contains a worst case scenario for every evacuation protocol. That is, for every  $\{\hat{P}, \hat{Y}\}$ , some  $s^* \in \mathcal{S}^*$  is a worst-case scenario for  $\{\hat{P}, \hat{Y}\}$ .

For  $k = 1$ , all the previous work on minmax regret on paths<sup>1</sup> used the existence of an  $O(n)$  size set that contains a worst-case scenario for each protocol. The intuitive reason why the algorithm in [30] blew up exponentially in  $k$  was that it needed to check an exponentially growing (in  $k$ ) set of worst case scenarios. The main observation of this paper, the one leading to improved algorithms, is that only a set of size  $O(n^2)$ , independent of  $k$ , is needed.

To derive  $\mathcal{S}^*$ , start by assuming that  $A$  has chosen  $\{\hat{P}, \hat{Y}\}$  and  $B$  has countered by choosing some *worst-case scenario*  $s^* \in \mathcal{S}$  for that  $\{\hat{P}, \hat{Y}\}$ . Let  $P_d \in \hat{P}$  be the dominant partition for  $\{\hat{P}, \hat{Y}\}$  under  $s^*$ .

Lemma 1 describes how to modify  $s^*$  outside  $P_d$ ; Theorem 1 describes how to modify  $s^*$  within  $P_d$ . Both types of modifications will maintain  $s^*$  as being worst case for  $\{\hat{P}, \hat{Y}\}$  and  $P_d$  as its dominant partition.

**Lemma 1** See Fig. 3a. Let  $s^* \in \mathcal{S}$  be a worst-case scenario for  $\{\hat{P}, \hat{Y}\}$  with  $P_d$  its associated dominant partition. Define  $s_B^*$  by

$$w_i(s_B^*) := \begin{cases} w_i(s^*) & \text{if } x_i \in P_d, \\ w_i^- & \text{if } x_i \notin P_d. \end{cases}$$

Then  $s_B^*$  is also a worst-case scenario for  $\{\hat{P}, \hat{Y}\}$  with dominant partition  $P_d$ .

**Proof** The evacuation time in  $P_d$  is the same for  $s_B^*$  as for  $s^*$  because  $w_i(s^*) = w_i(s_B^*)$  within  $P_d$ . The evacuation time in the other partitions might be less in  $s_B^*$  than  $s^*$  since

<sup>1</sup> On trees, a corresponding result for  $k = 1$  [9,21] shows the existence of a set of  $O(n^2)$  worst-case scenarios with nothing known when  $k > 1$ .



some of the values might be decreased but that can only reduce the evacuation times in *other* partitions. Thus

$$\Theta^k(P, \{\hat{P}, \hat{Y}\}, s_B^*) = \Theta^1(P_d, y_d, s_B^*) = \Theta^1(P_d, y_d, s^*) = \Theta^k(P, \{\hat{P}, \hat{Y}\}, s^*) \tag{7}$$

where  $y_d$  is the sink associated with  $P_d$ .

Furthermore, reducing the value of some  $w_i(s^*)$  can not increase the optimal evacuation time so,

$$\Theta_{\text{opt}}^k(P, s_B^*) \leq \Theta_{\text{opt}}^k(P, s^*). \tag{8}$$

Thus

$$\begin{aligned} R(\{\hat{P}, \hat{Y}\}, s_B^*) &= \Theta^k(P, \{\hat{P}, \hat{Y}\}, s_B^*) - \Theta_{\text{opt}}^k(P, s_B^*) \\ &\geq \Theta^k(P, \{\hat{P}, \hat{Y}\}, s^*) - \Theta_{\text{opt}}^k(P, s^*) \\ &= R(\{\hat{P}, \hat{Y}\}, s^*). \end{aligned}$$

Thus  $s_B^*$  is also a *worst-case scenario* with dominant partition  $P_d$ . □

Lemma 1 permits assuming that all vertices outside  $P_d$  have weights  $w_i^-$ . We now examine the scenario *inside*  $P_d$ .

**Definition 5** See Fig. 3b. Consider Partition  $P' = [x_l, x_r]$ . and let  $s \in \mathcal{S}$  be any fixed scenario.

A scenario  $s \in \mathcal{S}$  is called *left/right-dominant* for  $P'$  if,  $w_j(s) = w_j^-$  for  $j < l$  and  $j > r$  and there exists some  $i$  satisfying  $l \leq i \leq r$  such that

- *Left-dominant*:  $w_j(s) = w_j^+$  for  $l \leq j < i$  and  $w_j(s) = w_j^-$  for  $i \leq j \leq r$ .
- *Right-dominant*:  $w_j(s) = w_j^-$  for  $l \leq j < i$  and  $w_j(s) = w_j^+$  for  $i \leq j \leq r$ .

Given  $\hat{P}$  and  $P_i \in \hat{P}$ , let  $\mathcal{S}_L^i$  (resp.  $\mathcal{S}_R^i$ ) denote the set of all *left-dominant* (resp. *right-dominant*) scenarios for  $P_i$ .

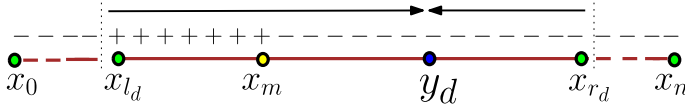
**Theorem 1** Fix  $\{\hat{P}, \hat{Y}\}$ . Let  $s^* \in \mathcal{S}$  be any *worst-case scenario* for  $\{\hat{P}, \hat{Y}\}$  and  $P_d$  its associated dominant partition. Then there exists a scenario  $s_B^* \in \mathcal{S}_L^d \cup \mathcal{S}_R^d$  such that  $s_B^*$  is also a *worst-case scenario* for  $\{\hat{P}, \hat{Y}\}$  with  $P_d$  its associated dominant partition.

**Proof** See Fig. 4. Let  $s^* \in \mathcal{S}$  be a *worst-case scenario* for  $\{\hat{P}, \hat{Y}\}$  with associated dominant partition  $P_d = [x_{l_d}, x_{r_d}] \in \hat{P}$ . Without loss of generality, from Lemma 1, we may assume that for  $x_i$  outside  $P_d$ ,  $w_i(s_B^*) = w_i^-$ .

To prove the Theorem we show that by only changing weights *within*  $P_d$ , we can modify  $s^*$  to  $s_B^* \in \mathcal{S}_L^d \cup \mathcal{S}_R^d$  while regret does not decrease.

Without loss of generality, assume that  $\Theta_L(P_d, y_d, s^*) \geq \Theta_R(P_d, y_d, s^*)$  (the other case is symmetric). From Eq. 1,

$$\Theta_L(P_d, y_d, s^*) = \max_{\substack{i : x_{l_d} \leq x_i < y_d \\ w_{d,i}^* > 0}} g(P_d, y_d, s^* : i) \tag{9}$$



**Fig. 4** Illustration of proof of Theorem 1. A left-dominant worst-case scenario for  $P_d$  with sink  $y_d$ . Left evacuation time dominates.  $m$  is the index  $i$  that maximizes Eq. 10

where

$$g(P_d, y_d, s^* : i) = (y_d - x_i)\tau + \frac{1}{c} W_{l_d, i}^{s^*}. \tag{10}$$

Let  $m$  be the index  $i$  which maximizes the right hand side of Eq. 9 (in case of ties, choose the smallest such  $i$ ).

We now show the transformation from  $s^*$  to  $s_B^*$ . There are three possible locations for vertices  $x_t \in P_d$  (see Fig. 4):

(i)  $y_d \leq x_t \leq x_{r_d}$  : For any such  $t$ , set  $w_t(s_B^*) = w_t^-$ . Reducing a weight can not increase evacuation time so

$$\Theta_R(P_d, y_d, s_B^*) \leq \Theta_R(P_d, y_d, s^*) \leq \Theta_L(P_d, y_d, s^*) = \Theta_L(P_d, y_d, s_B^*).$$

Thus  $\Theta^1(P_d, y_d, s_B^*) = \Theta^1(P_d, y_d, s^*)$ . By the definition of dominant partition,

$$\Theta^k(P, \{\hat{P}, \hat{Y}\}, s_B^*) = \Theta^1(P_d, y_d, s_B^*) = \Theta^1(P_d, y_d, s^*) = \Theta^k(P, \{\hat{P}, \hat{Y}\}, s^*).$$

Finally, reducing weights can only decrease the optimal evacuation time so

$$\Theta_{\text{opt}}^k(P, s_B^*) \leq \Theta_{\text{opt}}^k(P, s^*)$$

(ii)  $x_{m+1} \leq x_t < y_d$  : Again, for such  $t$ , set  $w_t(s_B^*) = w_t^-$ . After this transformation, for  $i \geq m + 1$ ,  $g(P_d, y_d, s_B^* : i) \leq g(P_d, y_d, s^* : i)$  while for  $i < m + 1$ ,  $g(P_d, y_d, s_B^* : i) = g(P_d, y_d, s^* : i)$  so, by the choice of  $m$ ,

$$\Theta_L(P_d, y_d, s^*) = \Theta_L(P_d, y_d, s_B^*)$$

and still  $\Theta^1(P_d, y_d, s_B^*) = \Theta^1(P_d, y_d, s)$ . By the same argument as in (i),

$$\Theta^k(P, \{\hat{P}, \hat{Y}\}, s_B^*) = \Theta^k(P, \{\hat{P}, \hat{Y}\}, s^*).$$

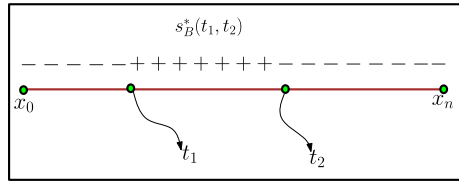
and

$$\Theta_{\text{opt}}^k(P, s_B^*) \leq \Theta_{\text{opt}}^k(P, s^*)$$

(iii)  $x_{l_d} \leq x_t \leq x_m$  : For such  $t$ , set  $w_t(s_B^*) = w_t^+$ . Then, simple calculation yields

$$\Theta_L(P_d, y_d, s_B^*) = \Theta_L(P_d, y_d, s^*) + (w_t^+ - w_t(s^*)) / c$$

**Fig. 5** Illustration of Definition 6. Scenario  $s_B^*(t_1, t_2)$  has  $w_i^+$  entries for  $t_1 \leq i < t_2$  and  $w_i^-$  entries everywhere else



where  $m$  remains the index that maximizes the value in Eq. 9. Thus,

$$\Theta^1(P_d, y_d, s_B^*) = \Theta_L(P_d, y_d, s_B^*) = \Theta_L(P_d, y_d, s^*) + (w_t^+ - w_t(s^*)) / c$$

so, again, by the definition of dominant partition,

$$\Theta^k(P, \{\hat{P}, \hat{Y}\}, s_B^*) = \Theta^k(P, \{\hat{P}, \hat{Y}\}, s^*) + (w_t^+ - w_t(s^*)) / c.$$

Also, increasing any one weight by can only increase the evacuation time by at most the amount of the increase divided by  $c$ , so

$$\Theta_{\text{opt}}^k(P, s_B^*) \leq \Theta_{\text{opt}}^k(P, s^*) + (w_t^+ - w_t(s^*)) / c$$

We have thus seen, for  $t$  in any of case (i), (ii), (iii),

$$\begin{aligned} R(\{\hat{P}, \hat{Y}\}, s_B^*) &= \Theta^k(P, \{\hat{P}, \hat{Y}\}, s_B^*) - \Theta_{\text{opt}}^k(P, s_B^*) \\ &\geq \Theta^k(P, \{\hat{P}, \hat{Y}\}, s^*) - \Theta_{\text{opt}}^k(P, s^*) \\ &= R(\{\hat{P}, \hat{Y}\}, s^*). \end{aligned}$$

$s^*$  is a worst case scenario with associated dominant partition  $P_d$ , so  $s_B^*$  must be one as well.

The discussion above considered changing only one  $w_t$ . Repeating this process for every  $t \in [x_{l_d}, x_{r_d}]$  yields a left dominant scenario,  $s_B^*$  for  $P_d$ . The case in which the right evacuation time dominates is totally symmetric and creates a right-dominant scenario for  $P_d$ .

We therefore can always create a worst case scenario in  $\mathcal{S}_L^d \cup \mathcal{S}_R^d$ . □

**Definition 6** See Fig. 5. For integers  $t_1, t_2$  satisfying  $0 \leq t_1 \leq t_2 \leq n$  define<sup>2</sup>  $s_B^*(t_1, t_2)$  as satisfying

$$w_i(s_B^*(t_1, t_2)) := \begin{cases} w_i^- & \text{if } 0 \leq i < t_1, \\ w_i^+ & \text{if } t_1 \leq i < t_2 \\ w_i^- & \text{if } t_2 \leq i \leq n \end{cases}$$

<sup>2</sup> Note that there is some redundancy in this definition in that,  $\forall t, s_B^*(t, t) = s_B^*(0, 0)$ .

and set

$$\mathcal{S}^* := \{s_B^*(t_1, t_2) : t_1, t_2, \text{ such that } 0 \leq t_1 \leq t_2 \leq n + 1\}$$

Note that all scenarios of the type described by Theorem 1 are in  $\mathcal{S}^*$ . Thus

**Property 2** Let  $\{\hat{P}, \hat{Y}\}$  be some evacuation protocol. Then there exists  $s_B^* \in \mathcal{S}^*$  that is a worst-case scenario for  $\{\hat{P}, \hat{Y}\}$ . Note that  $|\mathcal{S}^*| = O(n^2)$ .

If the  $\hat{P}$  is known in advance, there is a stronger property.

**Definition 7** Consider partition  $P_{lr} = [x_l, x_r]$ . Set

$$\mathcal{S}^*(P_{lr}) := \bigcup_{t=l}^r \{s_B^*(l, t), s_B^*(t, r + 1)\}.$$

Note that  $\mathcal{S}^*(P_{lr}) \subseteq \mathcal{S}^*$  and  $|\mathcal{S}^*(P_{lr})| = O(x_r - x_l + 1)$ .

Then Theorem 1 directly implies

**Property 3** Let  $\hat{P} = \{P_1, \dots, P_k\}$  fix  $\hat{Y}$  and suppose there exists some worst-case scenario  $s^*$  for  $\{\hat{P}, \hat{Y}\}$  with associated dominant partition  $P_d$ . Then there exists a scenario  $s_B^* \in \mathcal{S}^*(P_d)$  that is also a worst case scenario for  $\{\hat{P}, \hat{Y}\}$  with dominant partition  $P_d$ .

Next

**Definition 8** For  $\hat{P} = \{P_1, \dots, P_k\}$  set

$$\mathcal{S}^*(\hat{P}) := \bigcup_{1 \leq i \leq k} \mathcal{S}^*(P_i).$$

Note that  $\mathcal{S}^*(\hat{P}) \subseteq \mathcal{S}^*$  and  $|\mathcal{S}^*(\hat{P})| = \sum_i O(x_r - x_l + 1) = O(n)$ .

Since some partition must be dominant this yields

**Property 4** The set  $\mathcal{S}^*(\hat{P})$  includes some worst case scenario for  $\{\hat{P}, \hat{Y}\}$ .

### 4 Local Costs of Partitions

This section uses the properties derived in Sect. 3 to associate local costs with partitions and then show that the minmax regret value is the maximum of these local costs. This will enable designing efficient algorithms.

### 4.1 Local Partition Costs

**Theorem 2** Let  $l \leq r$ , set  $P_{lr} = [x_l, x_r]$  and let  $s \in \mathcal{S}$  be any scenario. Define

$$\forall y \in P_{lr}, \quad R_{lr}(y, s) := \Theta^1(P_{lr}, y, s) - \Theta_{\text{opt}}^k(P, s), \tag{11}$$

$$\forall y \in P_{lr}, \quad R_{lr}(y) := \max_{s \in \mathcal{S}^*} \{R_{lr}(y, s)\}, \tag{12}$$

and

$$R_{lr} := \min_{x_l \leq y \leq x_r} R_{lr}(y). \tag{13}$$

Then the minmax regret value for  $P$  satisfies

$$R_{\max}^k(P) = \min_{\hat{P}} \max_{1 \leq i \leq k} \{R_{lr_i}\} \tag{14}$$

where we denote  $\hat{P}$  by  $\hat{P} = \{P_1, \dots, P_k\}$  with  $P_i = [x_{l_i}, x_{r_i}]$

**Proof** The regret associated with  $\{\hat{P}, \hat{Y}\}$  under scenario  $s \in \mathcal{S}$  is

$$\begin{aligned} R(\{\hat{P}, \hat{Y}\}, s) &= \Theta^k(P, \{\hat{P}, \hat{Y}\}, s) - \Theta_{\text{opt}}^k(P, s) && \text{(from Eq. 5)} \\ &= \max_{1 \leq i \leq k} \left\{ \Theta^1(P_i, y_i, s) \right\} - \Theta_{\text{opt}}^k(P, s) && \text{(from Eq. 2)} \\ &= \max_{1 \leq i \leq k} \left\{ \Theta^1(P_i, y_i, s) - \Theta_{\text{opt}}^k(P, s) \right\} \\ &= \max_{1 \leq i \leq k} \left\{ R_{lr_i}(y_i, s) \right\}. \end{aligned} \tag{15}$$

The maximum in Eq. 15 is achieved when  $i = d$ , where  $P_d$  is the dominant partition.

The *max-regret* for  $\{\hat{P}, \hat{Y}\}$  can be written as:

$$\begin{aligned} R_{\max}(\{\hat{P}, \hat{Y}\}) &= \max_{s \in \mathcal{S}} \left\{ R(\{\hat{P}, \hat{Y}\}, s) \right\} && \text{(from Eq. 6)} \\ &= \max_{s \in \mathcal{S}^*} \left\{ R(\{\hat{P}, \hat{Y}\}, s) \right\} && \text{(from Prop 2)} \\ &= \max_{s \in \mathcal{S}^*} \max_{1 \leq i \leq k} \left\{ R_{lr_i}(y_i, s) \right\} && \text{(from Eq. 15)} \\ &= \max_{1 \leq i \leq k} \max_{s \in \mathcal{S}^*} \left\{ R_{lr_i}(y_i, s) \right\} \\ &= \max_{1 \leq i \leq k} \left\{ R_{lr_i}(y_i) \right\} && \text{(from Eq. 12)} \end{aligned} \tag{16}$$

**Table 1** The definitions of  $R_{lr}$ ,  $\bar{R}_{lr}$  and  $R'_{lr}$  and associated  $d$ ,  $\bar{d}$  and  $d'$  for given  $\hat{P}$

$R_{lr}(y) := \max_{s \in S^*} R_{lr}(y, s)$	$R_{lr} := \min_{x_l \leq y \leq x_r} R_{lr}(y)$	$d(\hat{P}) := \arg \max_{1 \leq i \leq k} R_{l_i r_i}$
	$\bar{R}_{lr} := \max\{R_{lr}, 0\}$	$\bar{d}(\hat{P}) := \arg \max_{1 \leq i \leq k} \bar{R}_{l_i r_i}$
$R'_{lr}(y) := \max_{s \in S^*(x_l, x_r)} R_{lr}(y, s)$	$R'_{lr} := \min_{x_l \leq y \leq x_r} R'_{lr}(y)$	$d'(\hat{P}) := \arg \max_{1 \leq i \leq k} R'_{l_i r_i}$

$S^*(x_l, x_r) \subseteq S^*$ . Lemma 6 shows that  $d = d'$ . Furthermore, if  $d > 0$  then  $d = d' = \bar{d}$

To complete the proof note that the minmax regret value can be written as:

$$\begin{aligned}
 R_{\max}^k(P) &= \min_{\{\hat{P}, \hat{Y}\}} R_{\max}(\{\hat{P}, \hat{Y}\}) \\
 &= \min_{\{\hat{P}, \hat{Y}\}} \max_{1 \leq i \leq k} \{R_{l_i r_i}(y_i)\} && \text{(from Eq. 16)} \\
 &= \min_{\hat{P}} \left\{ \min_{1 \leq i \leq k: y_i \in P_i} \left\{ \max_{1 \leq i \leq k} \{R_{l_i r_i}(y_i)\} \right\} \right\} \\
 &= \min_{\hat{P}} \left\{ \max_{1 \leq i \leq k} \left\{ \min_{1 \leq i \leq k: y_i \in P_i} R_{l_i r_i}(y_i) \right\} \right\} && (17) \\
 &= \min_{\hat{P}} \max_{1 \leq i \leq k} \{R_{l_i r_i}\}. && (18)
 \end{aligned}$$

□

Theorem 2 alone would permit developing a polynomial time Dynamic Programming algorithm for evaluating  $R_{\max}^k(P)$ . The remainder of this subsection defines two related quantities  $R'_{lr}$  and  $\bar{R}_{lr}$ , that permit developing faster algorithms. These are listed in Table 1 for comparison.

**Definition 9** Let  $l \leq r$ . Set

$$\begin{aligned}
 \text{(a)} \quad \forall y \in P_{lr}, \quad R'_{lr}(y) &:= \max_{s \in S^*(P_{lr})} \{R_{lr}(y, s)\}, \\
 R'_{lr} &:= \min_{x_l \leq y \leq x_r} R'_{lr}(y). && (19)
 \end{aligned}$$

$$\text{(b)} \quad \bar{R}_{lr} := \max\{R_{lr}, 0\} \tag{20}$$

We now prove that Eq. 14 will remain valid if  $R_{lr}$  is replaced by  $R'_{lr}$  or  $\bar{R}_{lr}$ . The final results that will be used for algorithmic development are stated in Lemma 6 and Theorem 3.

**Definition 10** Let  $\hat{P} = \{P_1, P_2, \dots, P_k\}$ . Set

$$\begin{aligned}
 d(\hat{P}) &:= \arg \max_{1 \leq i \leq k} R_{l_i r_i}, & d'(\hat{P}) &:= \arg \max_{1 \leq i \leq k} R'_{l_i r_i}, \\
 \bar{d}(\hat{P}) &:= \arg \max_{1 \leq i \leq k} \bar{R}_{l_i r_i}.
 \end{aligned}$$

**Lemma 2** Let  $d = d(\hat{P})$  where  $\hat{P} = \{P_1, P_2, \dots, P_k\}$ . Then

$$R_{l_d r_d} = \max_{1 \leq i \leq k} \{R_{l_i r_i}\} \geq 0. \tag{21}$$

**Proof** From Property 1, regret is always nonnegative, so  $R_{\max}^k(P) \geq 0$ . Thus, from Theorem 2,

$$0 \leq R_{\max}^k(P) = \min_{\hat{P}} \max_{1 \leq i \leq k} \{R_{l_i r_i}\}.$$

In particular, for any fixed  $\hat{P}$  satisfying  $d = d(\hat{P})$ , Eq. 21 holds. □

**Lemma 3**

$$\forall i, \quad \bar{R}_{ii} = 0 \tag{22}$$

**Proof** If  $y \in P_{ii}$  then  $y = x_i$ . Since for every scenario  $s$ ,  $\Theta^1(P_{ii}, x_i, s) = 0$ ,

$$R_{ii} = \max_{s \in S^*} \left( \Theta^1(P_{ii}, x_{ii}, s) - \Theta_{\text{opt}}^k(P, s) \right) = \max_{s \in S^*} \left( -\Theta_{\text{opt}}^k(P, s) \right) \leq 0,$$

so  $\bar{R}_{ii} = \max\{0, R_{ii}\} = 0$ . □

**Lemma 4**

$$\forall l \leq r, \quad R'_{lr} \leq R_{lr}.$$

**Proof** Recall that  $S^*(P_{lr}) \subset S^*$ , so

$$\forall y \in P_{lr}, \quad R'_{lr}(y) = \max_{s \in S^*(P_{lr})} R_{lr}(y, s) \leq \max_{s \in S^*} R_{lr}(y, s) = R_{lr}(y).$$

Thus

$$R'_{lr} = \min_{x_l \leq y \leq x_r} R'_{lr}(y) \leq \min_{x_l \leq y \leq x_r} R_{lr}(y) = R_{lr}. \tag{23}$$

□

**Lemma 5** Let  $d = d(\hat{P})$  where  $\hat{P} = \{P_1, P_2, \dots, P_k\}$ . Then

$$\forall y \in P_d, \quad R_{l_d, r_d}(y) = R'_{l_d, r_d}(y), \tag{24}$$

and

$$R_{l_d, r_d} = R'_{l_d, r_d}. \tag{25}$$

**Proof** For any  $y \in P_d$  set  $y_d = y$  and for  $i \neq d$  set  $y_i \in [x_l, x_{r_i}]$  to be a sink that achieves  $R_{l_i, r_i} = R_{l_i, r_i}(y_i)$ . Fix  $\hat{Y} = \{y_1, y_2, \dots, y_k\}$ .

Set  $\alpha = R_{l_d, r_d}$ . By the definition of  $d$ ,

$$\forall i < d, \max_{s \in S^*} R_{l_i r_i}(y_i, s) = R_{l_i r_i}(y_i) = R_{l_i r_i} < \alpha$$

and

$$\forall i > d, \max_{s \in S^*} R_{l_i r_i}(y_i, s) = R_{l_i r_i}(y_i) = R_{l_i r_i} \leq \alpha.$$

Furthermore

$$\max_{s \in S^*} R_{l_d r_d}(y_d, s) = R_{l_d r_d}(y_d) \geq \min_{l_d \leq y_d \leq r_d} R_{l_d r_d}(y_d) = \alpha.$$

Let  $s' \in S^*$  be such that  $R_{l_d r_d}(y_d, s') = R_{l_d r_d}(y_d)$ . Property 2 and the statements above imply that  $s'$  is a worst case scenario for  $\{\hat{P}, \hat{Y}\}$  and  $P_d$  is the dominant partition for  $\{\hat{P}, \hat{Y}\}$  under scenario  $s'$  (following Definition 2). Property 3 then implies the existence of  $s^* \in S^*(P_d)$  such that  $R_{l_d r_d}(y_d, s^*) = R_{l_d r_d}(y_d, s')$ . Since  $S^*(P_d) \subset S^*$ , this immediately implies

$$R'_{l_d, r_d}(y_d) = \max_{s \in S^*(P_d)} R_{l_d r_d}(y_d, s) = \max_{s \in S^*} R_{l_d r_d}(y_d, s) = R_{l_d, r_d}(y_d).$$

proving Eq. 24. Equation 25 follows from Eq. 24 and the definitions of  $R_{l_d, r_d}, R'_{l_d, r_d}$ :

$$R_{l_d, r_d} = \min_{x_{l_d} \leq y \leq x_{r_d}} R_{l_d, r_d}(y) = \min_{x_{l_d} \leq y \leq x_{r_d}} R'_{l_d, r_d}(y) = R'_{l_d, r_d}.$$

□

**Lemma 6** Fix  $\hat{P} = \{P_1, P_2, \dots, P_k\}$  with  $P_i = [x_{l_i}, x_{r_i}]$ . Set  $d = d(\hat{P}), d' = d'(\hat{P})$  and  $\bar{d} = \bar{d}(\hat{P})$ . Then

1.  $R_{l_d r_d} = R'_{l_{d'} r_{d'}} = \bar{R}_{l_{\bar{d}} r_{\bar{d}}}$ .
2. If  $R_{l_d r_d} = 0$ , then
  - $d = d'$
  - $\forall i, \bar{R}_{l_i r_i} = 0$ , so  $\bar{d} = 1$ .
3. If  $R_{l_d r_d} > 0$ , then  $d = d' = \bar{d}$ .

**Proof** From Lemma 2,

$$\max_{1 \leq i \leq k} \{\bar{R}_{l_i r_i}\} = \max_{1 \leq i \leq k} \{\max\{R_{l_i r_i}, 0\}\} = \max \left\{ \max_{1 \leq i \leq k} \{R_{l_i r_i}\}, 0 \right\} = \max_{1 \leq i \leq k} \{R_{l_i r_i}\}$$

and thus  $R_{l_d r_d} = \bar{R}_{l_{\bar{d}} r_{\bar{d}}}$ .



From Lemma 5,  $R_{l_{d'r_d}} = R'_{l_{d'r_d}}$ . From Lemma 4 and the definition of  $d'$ ,

$$R_{l_{d'r_d}} = R'_{l_{d'r_d}} \leq R'_{l_{d'r_{d'}}} = \max_{1 \leq i \leq k} \{R'_{l_i r_i}\} \leq \max_{1 \leq i \leq k} \{R_{l_i r_i}\} = R_{l_{d'r_d}}. \tag{26}$$

This implies that  $R'_{l_{d'r_d}} = R'_{l_{d'r_{d'}}} = R_{l_{d'r_d}}$  completing the proof of (1).

Furthermore, by the definition of  $d'$ ,  $d' \leq d$ . Suppose that  $d' < d$ . Then, again from Lemma 4, this would imply

$$R_{l_{d'r_d}} = R'_{l_{d'r_{d'}}} \leq R_{l_{d'r_{d'}}},$$

contradicting the definition of  $d$ . Thus  $d = d'$ ,

If  $R_{l_{d'r_d}} = 0$  then, from (1),  $\bar{R}_{\bar{d}r_{\bar{d}}} = 0$ . (2) then follows from the fact that

$$\forall i, \quad 0 \leq \bar{R}_{l_i r_i} \leq \bar{R}_{\bar{d}r_{\bar{d}}} = 0.$$

If  $0 < R_{l_{d'r_d}}$ , then, from (1)

$$\bar{R}_{l_{d'r_d}} = \max\{R_{l_{d'r_d}}, 0\} = R_{l_{d'r_d}} = \bar{R}_{\bar{d}r_{\bar{d}}}.$$

Recall that  $\forall j < d$ ,  $R_{l_j r_j} < R_{l_{d'r_d}}$ . Thus

$$\forall j < d, \quad \bar{R}_{l_j r_j} = \max(0, R_{l_j r_j}) < R_{l_{d'r_d}} = \bar{R}_{\bar{d}r_{\bar{d}}}$$

so  $\bar{d} = d$ . Combining with  $d = d'$  proves (3). □

Combining the above proves the main result.

**Theorem 3**

$$R^k_{\max}(P) = \min_{\hat{P}} \max_{1 \leq i \leq k} \{R_{l_i r_i}\} = \min_{\hat{P}} \max_{1 \leq i \leq k} \{\bar{R}_{l_i r_i}\} = \min_{\hat{P}} \max_{1 \leq i \leq k} \{R'_{l_i r_i}\}.$$

**Proof** Follows directly from Eq. 14 in Theorem 2 and part (1) in Lemma 6. □

The following observation will be quite useful in removing degenerate cases.

**Lemma 7** Fix  $\hat{P} = \{P_1, P_2, \dots, P_k\}$  and set  $d = d(\hat{P})$ . If  $R_{l_{d'r_d}} = 0$ ,  $R'_{l_{d'r_{d'}}} = 0$  or  $\bar{R}_{\bar{d}r_{\bar{d}}} = 0$  then  $R^k_{\max}(P) = 0$ .

**Proof** This follows directly from Lemma 6 and Theorem 3. □

### 4.2 Recurrence Relations for Minmax Regret

For  $\hat{P} = \{P_1, P_2, \dots, P_k\}$ , the values  $R_{l_i r_i}, \bar{R}_{l_i r_i}, R'_{l_i r_i}$  can be interpreted as (three different) *costs* of  $P_i = [x_{l_i}, x_{r_i}]$ . Solving for  $R_{\max}^k(P)$  using Theorem 3 is then the problem of finding the *minmax cost k-partition* with these  $P_i$  costs. Such partitioning problems can be solved in polynomial time using Dynamic Programming (DP). We encode the corresponding DP recurrences for costs  $\bar{R}_{l_r}$  and  $R_{l_r}$  in the following straightforward lemma without proof. Note that a corresponding equation also exists for the costs  $R_{l_r}$  but this would lead to a slower algorithm and is therefore not presented.

**Lemma 8** For  $0 \leq i \leq j \leq n$  and  $1 \leq q \leq k$  set

$$M_{ij}^q := \begin{cases} \text{undefined} & \text{if } j - i < q - 1, \\ \bar{R}_{ij} & \text{if } q = 1, \\ \min_{i \leq t < j} \max \{ \bar{R}_{it}, M_{(t+1)j}^{q-1} \} & \text{otherwise.} \end{cases} \tag{27}$$

$$M_{ij}'^q := \begin{cases} \text{undefined} & \text{if } j - i < q - 1, \\ R'_{ij} & \text{if } q = 1, \\ \min_{i \leq t < j} \max \{ R'_{it}, M_{(t+1)j}'^{q-1} \} & \text{otherwise.} \end{cases} \tag{28}$$

Then

$$M_{0n} = \min_{\hat{P}} \max_{1 \leq i \leq k} \{ \bar{R}_{l_i r_i} \} = R_{\max}^k(P) = \min_{\hat{P}} \max_{1 \leq i \leq k} \{ R'_{l_i r_i} \} = M'_{0n}.$$

If the  $\bar{R}_{ij}$  ( $R'_{ij}$ ) are precalculated, Eq. 27 (Eq. 28) defines a dynamic program for calculating all of the  $M_{ij}^q$  ( $M_{ij}'^q$ ) in  $O(n^3k)$  time.

For fixed  $i, j$ , from Definition 9(a), it is not difficult to see how to evaluate  $R'_{ij}$  in  $O(n^3)$  time (at least in the case when all sinks must be on vertices), thus calculating all the  $R'_{ij}$  in  $O(n^5)$  time. Once all  $R'_{ij}$  are known, Eq. 28 permits calculating  $M_{0n}^k = \min_{\hat{P}} \max_{1 \leq i \leq k} \{ R'_{l_i r_i} \} = R_{\max}^k(P)$  in  $O(n^3k)$  time, giving an  $O(n^5)$  algorithm.

We omit further details because we now present much faster algorithms for calculating  $M_{0n}^k$ . This will be efficient for calculating  $R_{\max}^k(P)$  for large  $k$ . Monotonicity as introduced in the next subsection will yield a better algorithm for calculating  $M_{0n}^k$ . This will be efficient for calculating  $R_{\max}^k(P)$  for small  $k$ .

### 4.3 Monotonic Sequences

The following property of the  $R_{l_r}$  will help speed up the calculations.

**Lemma 9** *Let  $l \leq r$ . Then*

- (a)  $R_{lr} \leq R_{l(r+1)}$ ,
- (b)  $R_{lr} \leq R_{(l-1)r}$ .

*Intuitively, increasing the size of a subpath can not decrease its regret.*

**Proof** We prove (a). The proof of (b) will be symmetric.

Let  $s$  be any scenario. and  $x_l \leq y \leq x_r$ . Extending  $P_{lr}$  one vertex to the right to create  $P_{l(r+1)}$  can not decrease evacuation time so

$$\Theta^1(P_{lr}, y, s) \leq \Theta^1(P_{l(r+1)}, y, s)$$

and then

$$\begin{aligned} R_{lr}(y, s) &= \Theta^1(P_{lr}, y, s) - \Theta_{\text{opt}}^k(P, s) \\ &\leq \Theta^1(P_{l(r+1)}, y, s) - \Theta_{\text{opt}}^k(P, s) = R_{l(r+1)}(y, s). \end{aligned}$$

Since this is true for every scenario  $s$ ,

$$R_{lr}(y) = \max_{s \in S^*} R_{lr}(y, s) \leq \max_{s \in S^*} R_{l(r+1)}(y, s) = R_{l(r+1)}(y) \quad (29)$$

so

$$R_{lr} = \min_{x_l \leq y \leq x_r} R_{lr}(y) \leq \min_{x_l \leq y \leq x_r} R_{l(r+1)}(y). \quad (30)$$

Now suppose  $x_r < y \leq x_{r+1}$ . Then, for every scenario  $s$ ,

$$\Theta^1(P_{lr}, x_r, s) = \Theta_L(P_{l(r+1)}, x_r, s) \leq \Theta_L(P_{l(r+1)}, y, s) \leq \Theta^1(P_{l(r+1)}, y, s)$$

and then

$$\begin{aligned} R_{lr}(x_r, s) &= \Theta^1(P_{lr}, x_r, s) - \Theta_{\text{opt}}^k(P, s) \\ &\leq \Theta^1(P_{l(r+1)}, y, s) - \Theta_{\text{opt}}^k(P, s) = R_{l(r+1)}(y, s). \end{aligned}$$

Continuing similar to above,

$$R_{lr}(x_r) = \max_{s \in S^*} R_{lr}(x_r, s) \leq \max_{s \in S^*} R_{l(r+1)}(y, s) = R_{l(r+1)}(y)$$

so

$$R_{lr} = \min_{x_l \leq y \leq x_r} R_{lr}(y) \leq R_{lr}(x_r) \leq \min_{x_r < y \leq x_{r+1}} R_{l(r+1)}(y) \quad (31)$$

Combining Eqs. (30) and (31) yields

$$\begin{aligned}
 R_{lr} &\leq \min \left( \min_{x_l \leq y \leq x_r} R_{l(r+1)}(y), \min_{x_r < y \leq x_{r+1}} R_{l(r+1)}(y) \right) \\
 &= \min_{x_l \leq y \leq x_{r+1}} R_{l(r+1)}(y) = R_{l(r+1)}
 \end{aligned}$$

proving (a). As noted, the proof of (b) is totally symmetric. □

We give this property a name and derive useful properties.

**Definition 11** Let  $A_{ij}$  be real values defined for all  $0 \leq i \leq j \leq n$ .  $A_{ij}$  is **monotonic** if  $A_{ii} = 0$  and

$$\forall 0 \leq i \leq j < n, \quad A_{ij} \leq A_{i(j+1)} \quad \text{and} \quad \forall 0 < i \leq j \leq n, \quad A_{ij} \leq A_{(i-1)j}.$$

The useful properties are

**Lemma 10** Let  $A_{ij}$  and  $B_{ij}$  be monotonic. For  $i \leq j$  define

$$C_{ij} := \begin{cases} 0 & \text{if } i = j \\ \min_{i \leq t < j} \max\{A_{it}, B_{(t+1)j}\} & \text{if } i < j \end{cases}$$

1. Then  $C_{ij}$  is monotonic.
2. For  $j > i$  set  $t[i, j] = \min\{t : i \leq t < j \text{ and } A_{it} \geq B_{(t+1)j}\}$ .

(Note: such a  $t$  must exist because  $A_{i(j-1)} \geq 0 = B_{jj}$ .) Then

$$\forall i \in [i, j], \quad A_{it} < B_{(t+1)j} \quad \text{if and only if} \quad t < t[i, j], \tag{32}$$

$$C_{ij} = \min\{A_{it[i,j]}, B_{t[i,j]j}\} \tag{33}$$

3.  $\forall 1 \leq i < j < n, \quad t[i, j] \leq t[i, j + 1]$

**Proof** 1. From the monotonicity of  $B_{ij}$ ,

$$\forall t, \quad i \leq t < j, \quad B_{(t+1)j} \leq B_{(t+1)(j+1)}$$

so

$$\max\{A_{it}, B_{(t+1)j}\} \leq \max\{A_{it}, B_{(t+1)(j+1)}\}.$$

From the monotonicity of  $A_{ij}$  and the fact  $B_{jj} = B_{(j+1)(j+1)} = 0$ ,

$$\max\{A_{i(j-1)}, B_{jj}\} = A_{i(j-1)} \leq A_{ij} = \max\{A_{ij}, B_{(j+1)(j+1)}\}.$$

Thus

$$C_{ij} = \min_{i \leq t < j} \max\{A_{it}, B_{(t+1)j}\} \leq \min_{i \leq t < j+1} \max\{A_{it}, B_{(t+1)(j+1)}\} = C_{i(j+1)}.$$

The proof that  $C_{i,j} \leq C_{(i-1)j}$  is symmetric.

2. If  $j = i + 1$  then  $t[i, j] = i$  and Eqs. 32 and 33 are trivially satisfied with  $C_{ij} = \max\{A_{ii}, B_{jj}\} = 0 = \min\{A_{ii}, B_{ij}\}$ .  
 Otherwise,  $j > i + 1$ . From the definition of  $t[i, j]$ ,  $\forall i \leq t < t[i, j]$ ,  $A_{it} \leq B_{(t+1)j}$ . Next, by monotonicity, and the definition of  $t[i, j]$ ,

$$\forall t[i, j] \leq t < j, \quad A_{it} \geq A_{it[i,j]} \geq B_{(t[i,j]+1)j} \geq B_{(t+1)j}$$

proving Eq. 32. To prove Eq. 33 note that by monotonicity,

$$\min_{i \leq t < t[i,j]} \max\{A_{it}, B_{(t+1)j}\} = \min_{i \leq t < t[i,j]} B_{(t+1)j} = B_{t[i,j]j}$$

and

$$\min_{t[i,j] \leq t < j} \max\{A_{it}, B_{(t+1)j}\} = \min_{t[i,j] \leq t < j} A_{i,t} = A_{i,t[i,j]}$$

3. Note that, if  $A_{it} < B_{(t+1)j}$  then, by monotonicity,  $A_{it} \leq B_{(t+1)j} \leq B_{(t+1)(j+1)}$  and the proof follows from part (2). □

Monotonic sequences are relevant because Lemmas 3 (Eq. 22) and 9 imply that the  $\bar{R}_{ij}$  are monotonic and thus from Lemma 10(1)

**Corollary 1** *For every fixed  $q$ ,  $0 \leq q \leq k$  the  $M_{i,j}^q$  defined in Lemma 8 are monotonic.*

*Note* For later use we note that Lemma 9 and Corollary 1 no longer hold if  $R_{lr}$  is replaced by  $R'_{lr}$ . This is because, after this substitution, Eqs. 29 and 30 no longer remain valid because  $S^*(P_{lr}) \not\subseteq S^*(P_{l(r+1)})$ , so it is not necessarily true that  $\max_{s \in S^*(P_{lr})} R_{lr}(s, y) \leq \max_{s \in S^*(P_{l(r+1)})} R_{l(r+1)}(s, y)$ . Thus, the  $R'_{lr}$  are not necessarily monotonic. This fact will have later implications in the algorithmic design.

### 5 $\Theta^k(P, x, s)$ and $R_{ij}$ : Properties and Calculation

Equations 27 and 28 lead directly to two different dynamic programming solutions for calculating the minmax regret; one based on  $\bar{R}_{ij}$  and the other on  $R'_{ij}$ . Using them requires efficient calculation of  $R_{ij}$  and  $R'_{ij}$  which first requires efficient calculation of  $\Theta^k(P, s)$ .

Earlier papers, e.g., [9,16,20,27,34], in this area have already developed techniques for fast calculation of  $\Theta^k(P, s)$  for fixed  $s$ . The closest to our needs is the *Critical Vertex Tree* data structure of [9]. A straightforward modification of [9]’s design yields the following:

**Lemma 11** *The Critical Vertex Tree (CVT) Data Structure can be built in  $O(n)$  time and, once built, permits the following operations:*

*Let  $i \leq j$  and set  $P_{ij} = [x_i, x_j]$ . Let  $0 \leq t_1 \leq t_2 \leq n$  and set  $s = s_B^*(t_1, t_2)$  as introduced in Definition 6. Then all of the below can be calculated in  $O(\log n)$  time.*

1.  $\Theta_L(P_{ij}, x, s)$  and  $\Theta_R(P_{ij}, x, s)$  for any  $x \in P_{ij}$ .
2.  $\Theta^1(P_{ij}, s)$  and the value  $x^*$  such that  $\Theta^1(P_{ij}, s) = \max \{ \Theta_L(P_{ij}, x^*, s), \Theta_R(P_{ij}, x^*, s) \}$ .
3. For any  $\alpha > 0$ ,  $\max\{x' \in P_{ij} : \Theta_L(P_{ij}, x', s) \leq \alpha\}$ .
4. For any  $x \in P_{ij}$  and  $\alpha > 0$ ,  $\max\{j' \leq j : \Theta_R([x, x'], x, s) \leq \alpha\}$ .

We note that [9] defines the Critical Vertex Tree for a fixed scenario  $s$ , provides the  $O(n)$  construction algorithm, proves point (1) and gives an  $O(\log^2 n)$  (and not  $O(\log n)$ ) algorithm for (2). In the ‘‘Appendix’’ (Sect. 8) we describe the modifications required to extend their construction to yield the full strength of Lemma 11.

The remainder of this section, though, assumes the correctness of Lemma 11 and describes how it permits fast construction of  $R_{ij}$  and  $R'_{ij}$ . The first step is to show that it permits fast calculation of  $\Theta_{\text{opt}}^k(P, s)$  for  $s \in S^*$ .

For simplification, the derivations often use the following substitution.

**Definition 12** Let  $s$  be fixed. For  $i \leq j$ , define

$$A_{ij}^q := \Theta_{\text{opt}}^q(P_{ij}, s).$$

**Lemma 12** For fixed  $s$  and all  $q$ ,  $A_{ij}^q$  is a monotonic sequence.

**Proof** Adding a vertex to a subpath can only increase its 1-sink evacuation time so  $A_{ij}^1 \leq A_{i(j+1)}^1$  and  $A_{(i-1)j}^1 \leq A_{ij}^1$ . Since  $A_{ii}^1 = 0$ , the sequence  $A_{ij}^1$  is monotonic.

The proof that  $A_{ij}^q$  is monotonic will be by induction on  $q$ . Assume that  $A_{ij}^{q-1}$  is monotonic (as has just been shown for  $q = 2$ ). Now note that

$$\begin{aligned} A_{ij}^q &= \Theta_{\text{opt}}^q(P_{ij}, s) = \min_{i \leq t < j} \max\{\Theta_{\text{opt}}^1(P_{it}, s), \Theta_{\text{opt}}^{q-1}(P_{(t+1)j}, s)\} \\ &= \min_{i \leq t < j} \max\{A_{it}^1, A_{(t+1)j}^{q-1}\}, \end{aligned}$$

so by Lemma 10,  $A_{ij}^q$  is also monotonic. □

The algorithm for calculating the optimum value  $\Theta_{\text{opt}}^q(P_{ij}, s)$  will utilize the following feasibility test as a subroutine.

**Lemma 13** Assume  $i \leq j$ ,  $\alpha \geq 0$  and  $s = s_B^*(t_1, t_2)$ . Define

$$Feasible(q : i, j, \alpha, s) := \begin{cases} \text{TRUE,} & \text{if } \Theta_{\text{opt}}^q(P_{ij}, s) \leq \alpha, \\ \text{FALSE,} & \text{if } \Theta_{\text{opt}}^q(P_{ij}, s) > \alpha. \end{cases}$$

Assuming a pre-constructed CVT,  $Feasible(q : i, j, \alpha, s)$  can be calculated in  $O(q \log n)$  time.

**Proof** Note that if  $i = j$ , then  $A_{ij}^q = \Theta_{\text{opt}}^1(P_{ii}, s) = 0$ , so  $Feasible(q : i, j, \alpha, s) = \text{TRUE}$ .

If  $i < j$  and  $q = 1$  :

$A_{ij}^1 = \Theta_{\text{opt}}^1(P_{ij}, s) \leq \alpha$  can be checked in  $O(\log n)$  time by using Lemma 11 to directly calculate  $\Theta_{\text{opt}}^1(P_{ij}, s)$ .

If  $i < j$  and  $q > 1$  :

Set

$$t := \max\{t' : t' \leq j \text{ and } \Theta_{\text{opt}}^1(P_{it'}, s) \leq \alpha\}.$$

This can be calculated in  $O(\log n)$  time by first using Lemma 11(3) to find

$$x'' = \max\{x' \in P_{ij} : \Theta_L(P_{ij}, x', s) \leq \alpha\}$$

and then using Lemma 11(4) to find

$$t = \max\{j' : j' \leq j \text{ and } \Theta_R(P_{ij'}, x'', s) \leq \alpha\}.$$

If  $t = j$  then

$$A_{it}^q = \Theta_{\text{opt}}^q(P_{it}, s) \leq \Theta_{\text{opt}}^1(P_{it}, s) \leq \alpha$$

so  $Feasible(q : i, j, \alpha, s) = \text{TRUE}$ . If  $t < j$ , then  $A_{it}^1 \leq \alpha$  and  $A_{i(t+1)}^1 > \alpha$ . From the monotonicity of  $A_{i,j}^q$  in Lemma 12,

$$A_{ij}^q \leq \alpha \quad \text{if and only if} \quad A_{(t+1)j}^{q-1} \leq \alpha.$$

So, if  $t < j$ ,

$$Feasible(q : i, j, \alpha, s) = Feasible(q - 1 : t + 1, j, \alpha, s). \tag{34}$$

Thus  $Feasible(q : i, j, \alpha, s)$  can be implemented via a recursive procedure that does  $O(\log n)$  work to calculate  $t$  and then calls  $Feasible(q - 1 : t + 1, j, \alpha, s)$ . When  $q = 1$  the evaluation can be done in  $O(\log n)$  time, so the full evaluation of  $Feasible(q : i, j, \alpha, s)$  can be implemented in  $O(q \log n)$  time as required.  $\square$

**Lemma 14** Let  $s = s_B^*(t_1, t_2)$  and  $i < j$ . Assuming a pre-constructed CVT,  $\Theta_{\text{opt}}^k(P_{ij}, s)$  can be calculated in  $O(k^2 \log^2 n)$  time.

**Proof** Let  $q \leq k$ . Set

$$t_q[i, j] = \min\{t : i \leq t < j \text{ and } A_{it}^1 \geq A_{(t+1)j}^{q-1}\}.$$

Lemmas 10 and 12 guarantee that, if  $t \in [i, j]$  then

$$i \leq t < t_q[i, j] \iff A_{it}^1 < A_{(t+1)j}^{q-1}, \tag{35}$$

and

$$A_{ij}^q = \min \left\{ A_{it_q[i,j]}^1, A_{t_q[i,j]j}^{q-1} \right\}. \tag{36}$$

Equation 35 implies that  $t_q[i, j]$  can be found via binary search using  $O(\log n)$  queries of the form  $(t < t_q[i, j]?)$  which are equivalent to the queries  $(A_{it}^1 < A_{(t+1)j}^{q-1}?)$ . This can be implemented by

1. Calculating  $\alpha = A_{it}^1 = \Theta_{\text{opt}}^1(P_{it}, s)$  in  $O(\log n)$  time using Lemma 11 (1);
2. then noting that  $A_{it}^1 < A_{(t+1)j}^{q-1}$  iff  $Feasible(q - 1 : t + 1, j, \alpha, s) = \text{FALSE}$ .

From Lemma 13, the time for one query is  $O(q \log n)$  so the total binary search time to find  $t_q[i, j]$  is  $O(q \log^2 n)$ . The full algorithm<sup>3</sup> to calculate  $A_{ij}^q$  is then

1. If  $q = 1$ , calculate  $A_{ij}^1$  in  $O(\log n)$  time using Lemma 11 (1)
2. Else if  $q > 1$
3. Find  $t_q[i, j]$  in  $O(q \log^2 n)$  time
4. Calculate  $A_{it_q[i,j]}^1$  in  $O(\log n)$  time using Lemma 11 (1)
5. Recursively calculate  $A_{t_q[i,j]j}^{q-1}$
6. Return  $\min \left\{ A_{it_q[i,j]}^1, A_{t_q[i,j]j}^{q-1} \right\}$ .

Let  $F_q(n)$  be the time required to calculate  $A_{ij}^q$  where  $n = j - i + 1$ . The recurrence relation is then

$$F_q(n) = \begin{cases} O(\log n) & \text{if } q = 1, \\ F_{q-1}(n) + O(q \log^2 n) & \text{if } q > 1, \end{cases}$$

which solves out to  $F_k(n) = O(k^2 \log^2 n)$ . □

### 5.1 Evacuation Costs for Sinks not on Vertices

The following simple observation about the linearity of evacuation as the sink moves along an edge *between* two vertices will be needed. It follows directly from the evacuation cost formulas in Eqs. 1 and 2:

**Lemma 15** *Suppose  $i < k \leq j$  and  $x \in P_{(k-1)k}$ . For any scenario  $s$ , if  $x_{k-1} < x \leq x_k$ , then*

$$\Theta_L(P_{ij}, x, s) = \Theta_L(P_{ik}, x_k, s) - \tau(x_k - x) = \Theta_L(P_{ik}, s) - \tau(x_k - x)$$

and if  $x_{k-1} \leq x < x_k$ ,

---

<sup>3</sup> Without delving into details we note that the algorithm presented can alternatively be derived as a *parametric search* [29] implementation of binary search using the feasibility test of Lemma 13 to implement the binary search comparison.



$$\begin{aligned}\Theta_R(P_{ij}, x, s) &= \Theta_R(P_{(k-1)j}, x_{k-1}, s) - \tau(x - x_{k-1}) \\ &= \Theta_R(P_{(k-1)j}, s) - \tau(x - x_{k-1}).\end{aligned}$$

$\Theta_L(P_{ij}, x_k, s)$  is the time required for the last piece of flow from within  $P_{i(k-1)}$  to reach  $x_k$ . The intuition for the first equation is that no congestion can occur *inside* an edge so the last flow reached  $x$  exactly  $\tau(x_k - x)$  before it reached  $x_k$ . The intuition for the second equation is similar.

## 5.2 Strict Unimodality and Its Applications

**Definition 13** Let  $f(x)$  be a function on a real interval  $I = [y, z]$ .  $f(x)$  is *strictly unimodal* if there exists some  $x^* \in I$  such that  $f(x)$  is monotonically decreasing in  $[x, x^*]$  and monotonically increasing in  $[x^*, y]$ . It is not necessary that  $f(x)$  be continuous. Note that  $x^*$  is the unique minimum location of  $f(x)$ .

Strictly unimodal functions arise quite naturally. The proof of the following lemma is straightforward and therefore omitted.

**Lemma 16** Let  $f(x)$ ,  $g(x)$  be, respectively, monotonically increasing and decreasing functions on  $I$ . Then  $h(x) = \max\{f(x), g(x)\}$  is a strictly unimodal function on  $I$ .

Finally, the maximum of strictly unimodal functions is easily shown to be strictly unimodal.

**Lemma 17** If  $f_1(x)$  and  $f_2(x)$  are both strictly unimodal on  $I$  then  $f(x) = \max\{f_1(x), f_2(x)\}$  is also strictly unimodal on  $I$ .

More generally, if  $f_i(x)$ ,  $i = 1, 2, \dots, j$  are all strictly unimodal on  $I$  then  $f(x) = \max_{1 \leq i \leq j} f_i(x)$  is also strictly unimodal on  $I$ .

Strictly unimodal functions arise in our problem due to the following:

**Lemma 18** Let  $i \leq j$  and  $s \in S$ . Then

1.  $\Theta^1(P, x, s)$  is a strictly unimodal function on  $P$  as a function of  $x$ .
2.  $R_{ij}(y, s)$  is a strictly unimodal function on  $P_{ij}$  as a function of  $y$ .
3.  $R_{ij}(y)$  is a strictly unimodal function on  $P_{ij}$  as a function of  $y$ .
4.  $R'_{ij}(y)$  is a strictly unimodal function on  $P_{ij}$  as a function of  $y$ .

**Proof** 1. This follows directly from the definition

$$\Theta^1(P, x, s) = \max\{\Theta_L(P, x, s), \Theta_R(P, x, s)\}, \quad (37)$$

$\Theta_L(P, x, s)$  being monotonically increasing in  $x$ ,  $\Theta_R(P, x, s)$  being monotonically increasing in  $x$  and Lemma 16.

2. From the definition

$$R_{ij}(s, y) = \Theta^1(P_{ij}, y, s) - \Theta_{\text{opt}}^k(P, s).$$

Part (1) states that  $\Theta^1(P_{ij}, y, s)$  is strictly unimodal. Subtracting a constant from a strictly unimodal function leaves another strictly unimodal function.

3. Apply Part (2) and Lemma 17 to

$$R_{ij}(y) = \max_{s \in S^*} \{R_{ij}(y, s)\}.$$

4. Again apply Part (2) and Lemma 17, but this time to

$$R'_{ij}(y) = \max_{s \in S^*(P_{ij})} \{R_{ij}(y, s)\}.$$

□

Binary searching on Strictly Unimodal functions is known to be “easy”.

**Lemma 19** *Let  $f(x)$  be a strictly unimodal function defined in  $[x, y]$  and  $x^*$  the location of its unique minimum value. Let  $x = x_1 < x_2 < \dots < x_n = y$  be a sequence of points and  $g(n)$  the time required to evaluate  $g(x_i)$  for any  $i$ . Then, in  $O(g(n) \log n)$  time, binary search can determine the index*

$$t = \max\{i : 1 \leq i \leq n, \text{ and } x_i \leq x^*\}$$

Note that if  $t = n$  then  $x^* = x_n = y$ . Otherwise  $x_t \leq x^* < x_{t+1}$ .

This immediately leads to the major technical construction lemma

**Lemma 20** *Let  $i \leq j$  be integers,  $s \in S^*$  a scenario,  $x \in P_{ij}$ ,*

$$\begin{aligned} \alpha(n) &= \text{Time needed to evaluate } \Theta_{\text{opt}}^k(P, s), \\ \beta(i, j, s) &= \text{Time needed to evaluate } \Theta_L(P_{ij}, x, s) \text{ and } \Theta_R(P_{ij}, x, s). \end{aligned}$$

Recall that

$$(a) R_{ij} = \min_{x_i \leq y \leq x_j} R_{ij}(y) \text{ and } (b) R'_{ij} = \min_{x_i \leq y \leq x_j} R'_{ij}(y). \tag{38}$$

Then

(a)  $R_{ij}$  and the  $y$  at which the minimum is achieved in Eq. 38 (a) can be evaluated in time

$$O(n^2\alpha(n) + n^2\beta(i, j, s) \log n).$$

(b)  $R'_{ij}$  and the  $y$  at which the minimum is achieved in Eq. 38 (b) can be evaluated in time

$$O((j - i + 1)\alpha(n) + (j - i + 1)\beta(i, j, s) \log n).$$

**Proof** (a) Use  $O(n^2\alpha(n))$  time to calculate  $\Theta_{\text{opt}}^k(P, s)$  for all the  $O(n^2)$  scenarios  $s \in S^*$ . Store these  $O(n^2)$  values so that they can be retrieved in  $O(1)$  time.

For fixed  $s$  and  $y$ , calculating

$$R_{ij}(y, s) = \max \{ \Theta_L(P_{ij}, y, s), \Theta_R(P_{ij}, y, s) \} - \Theta_{\text{opt}}^k(P, s)$$

only requires an additional  $O(\beta(i, j, s))$  time.

$R_{ij}(y)$  can then be evaluated in  $O(n^2\beta(i, j, s))$  time by evaluating  $R_{ij}(y, s)$  at each of the  $O(n^2)$   $s \in S^*$  and returning the maximum. Call such a calculation of  $R_{ij}(y)$  a “query”.

Since  $R_{ij}(y)$  is strictly unimodal as a function of  $y$ , there exists a unique location  $x^*$  at which it achieves its minimum.

Using  $O(\log n)$  queries, Lemma 19 finds either that  $x^* = x_j$  or returns a  $t$  such that  $x^* \in [x_t, x_{t+1})$ .

In the first case, the argument above permits calculating  $R_{ij}(x^*) = R_{ij}(x_j)$  in  $O(n^2\beta(i, j, s))$  time.

In the second case, the same argument permits calculating  $R_{ij}(x_t)$  in  $O(n^2\beta(i, j, s))$  time.

If  $x^* \neq x_t$  then,  $x^* = x_t + \delta$  for some  $\delta > 0$  and, from Lemma 15,

$$\begin{aligned} \Theta_R(P_{ij}, x_t + \delta, s) &= \Theta_R(P_{ij}, x_t, s) - \tau(x^* - x_t) \\ \Theta_L(P_{ij}, x_t + \delta, s) &= \Theta_L(P_{ij}, x_{t+1}, s) - \tau(x_{t+1} - x^*) \end{aligned}$$

Plugging in the definitions and collecting terms yields,  $\forall x \in (x_t, x_{t+1})$ ,

$$\begin{aligned} R_{ij}(x, s) &= \max \left\{ \Theta_R(P_{ij}, x_t, s) - \Theta_{\text{opt}}^k(P, s) - \tau(x - x_t), \right. \\ &\quad \left. \Theta_R(P_{ij}, x_{t+1}, s) - \Theta_{\text{opt}}^k(P, s) - \tau(x_{t+1} - x) \right\} \\ &= \max \left\{ A_s - \tau x, B_s + \tau x \right\} \end{aligned}$$

where  $A_s$  and  $B_s$  are appropriately defined constants. Thus  $\forall x \in (x_t, x_{t+1})$ ,

$$\begin{aligned} R_{ij}(x) &= \max_{s \in S^*} \{ R_{ij}(x, s) \} \\ &= \max_{s \in S^*} \max \{ A_s - \tau x, B_s + \tau x \} \\ &= \max \left\{ \max_{s \in S^*} \{ A_s - \tau x \}, \max_{s \in S^*} \{ B_s + \tau x \} \right\} \\ &= \max \left\{ \left( \max_{s \in S^*} A_s \right) - \tau x, \left( \max_{s \in S^*} B_s \right) + \tau x \right\} \\ &= \max \{ A - \tau x, B + \tau x \} \end{aligned}$$

where  $A = \max_{s \in S^*} A_s, B = \max_{s \in S^*} B_s$ . If the  $A_s, B_s$  and thus the  $A, B$  were known then, by linearity, finding the unique  $x^* \in [x_t, x_{t+1})$  such that

$$R_{i,j}(x^*) = \min_{x \in [x_t, x_{t+1})} R_{ij}(x) = \min \left( R_{ij}(x_t), \min_{x \in (x_t, x_{t+1})} R_{ij}(x) \right)$$

needs only  $O(1)$  time. Since  $A_s$  and  $B_s$  can be calculated in a further  $O(|S^*|(\alpha(n) + \beta(i, j, s)))$  time where  $|S^*| = O(n^2)$ , the proof is complete.

(b) Start by spending  $O((j - i + 1)\alpha(n))$  time calculating  $\Theta_{\text{opt}}^k(P, s)$  for all the  $O((j - i + 1))$  scenarios  $s \in S^*(P_{ij})$ . The evaluation of  $R'_{ij}$  is exactly the same as that of  $R'_{ij}$  in (a) except that  $s \in S^*$  is replaced by  $s \in S^*(P_{ij})$ . Since  $|S^*(P_{ij})| = O(j - i + 1)$  this would just replace every  $n^2$  in the analysis with  $|j - i + 1|$  which is the claimed result.  $\square$

### 5.3 Quick Construction of $R_{ij}$ and $R'_{ij}$

The tools just developed provide quick proofs of the following two lemmas.

**Lemma 21** *Assume a pre-constructed CVT. For any  $i < j$ ,  $R_{ij}$  can be evaluated in  $O(n^2 k^2 \log^2 n)$  time and can be evaluated in  $R'_{ij}$  in  $O(|j - i + 1| k^2 \log^2 n)$  time.*

**Proof** Lemma 11 immediately provides that  $\forall s \in S^*, \beta(i, j, s) = O(\log n)$ .

Lemma 14 gives  $\alpha(n) = O(k^2 \log^2 n)$ .

Plugging back into Lemma 20 completes the proof.  $\square$

**Lemma 22** *All  $O(n^2) R'_{ij}$  can be constructed in  $O(n^3 \log n)$  total time.*

**Proof** First use  $O(n)$  time to construct the CVT.

Next calculate  $\Theta_{\text{opt}}^k(P, s)$  for each of the  $O(n^2) s \in S^*$ . As previously mentioned, [7] describes how to calculate one  $\Theta_{\text{opt}}^k(P, s)$  in  $O(n \log n)$  time so this takes  $O(n^3 \log n)$  time. Store the values in an array so each can be accessed in  $O(1)$  time.

For fixed  $i, j$ , Lemma 11 permits calculating  $\Theta_L(P_{ij}, x_j, s)$  and  $\Theta_R(P_{ij}, x_i, s)$  in  $O(\log n)$  time. Use  $O(n^2 \log n)$  time<sup>4</sup> to calculate these for all pairs  $i, j$  and then store the  $O(n^2)$  values in an array so that they can be accessed in  $O(1)$  time.

Then apply Lemma 20 with  $\alpha(n) = O(1)$  and  $\beta(i, j, s) = O(1)$  to see that we can calculate  $R'_{ij}$  in  $O(|j - i + 1| \log n) = O(n \log n)$  time. Thus, all  $O(n^2) R'_{ij}$  can be calculated in  $O(n^3 \log n)$  time.  $\square$

## 6 Algorithm Design

The tools developed now permit designing efficient algorithms for calculating the minmax regret value  $R_{\text{max}}^k(P)$  and the associated  $\{\hat{P}^*, \hat{Y}^*\}$  that achieve that value.

<sup>4</sup> We note that a more delicate algorithm could actually calculate all  $O(n^2)$  values in  $O(n^2)$  time from scratch. That would not improve the overall running time, though, so we do not provide details.

Section 6.1 provides a simple dynamic programming algorithm based on Eq. 27 and  $R'_{ij}$  that runs in  $O(n^3 \log n)$  time, independent of  $k$ .

Section 6.2 combines Eq. 28 and the monotonicity of the  $R_{ij}$  to develop a  $O(n^2 k^2 \log^{k+1} n)$  time nested binary-search algorithm.

If the  $R'_{ij}$  were also monotonic, then directly replacing the  $R_{ij}$  in Sect. 6.2 with  $R'_{ij}$  would immediately lead to an improved  $O(nk^2 \log^{k+1} n)$  algorithm. Unfortunately, this is not necessarily true, so that approach fails. The relationship between the  $R_{ij}$  and  $R'_{ij}$  derived in Lemma 6, though, will permit modifying the algorithm in Sect. 6.2 to run in  $O(nk^2 \log^{k+1} n)$  by *simulating* sequences of calls to  $R_{ij}$  with calls to  $R'_{ij}$ .

This modification is first illustrated for the simpler  $k = 2$  case in Sect. 6.3 and then generalized to all  $k$  in Sect. 6.4.

The algorithm for  $k = 2$  runs in  $O(n \log^3 n)$ , improving the previously best known  $O(n \log^4 n)$  algorithm of [9]. To put this into context we note that the algorithm of [9] was actually based on the properties in Sects. 3 and 4 as quoted from an unpublished earlier version [2] of this paper combined with their earlier version of the CVT. Unwinding [9]’s algorithm, it can be seen that the  $O(\log n)$  improvement for the  $k = 2$  case primarily comes from the  $O(\log n)$  improvement to the CVT in our Lemma 11 (2).

### 6.1 A Simple Dynamic Program

For  $q = 0, 1, \dots, k$  set  $M'(q : i) := M'^q_{in}$  as defined in Lemma 8. Then

$$M'(q : i) = \begin{cases} \text{undefined} & \text{if } q > 1 \text{ and } n - i < q - 1 \\ R'_{in} & \text{if } q = 1 \\ \min_{i \leq t < n} \max \{R'_{it}, M'(q - 1 : t + 1)\} & \text{otherwise} \end{cases} \tag{39}$$

From Theorem 3,  $M'(k : 0) = M^k_{0n} = R^k_{\max}(P)$ . Use Lemma 22 to calculate all of the  $R'_{i,j}$  values in  $O(n^3 \log n)$  time, and then DP Eq. 39 to fill in all of the  $M'(q : i)$  table entries in  $O(n^2 k)$  time ( $O(n)$  time per entry) leading to

**Theorem 4** For all  $k \leq n$ ,

$$R^k_{\max}(P) = \min_{\{\hat{P}, \hat{Y}\}} R_{\max}(\{\hat{P}, \hat{Y}\}) = M^k_{0n}$$

can be calculated in  $O(n^3 \log n)$  time.

Note that this algorithm actually calculates the values  $R^k_{\max}(P)$  for all  $k \leq n$ , not just for one specific  $k$ . Also, the dynamic program can be unwound using backtracking to find the  $\{\hat{P}^*, \hat{Y}^*\}$  that achieves the minimum value and the worst case scenario associated with it.

### 6.2 A First Binary Search Based Algorithm

For  $q = 0, 1, \dots, k$ , set  $M(q : i) := M_{in}^q$  as defined in Eq. 28. Then

$$M(q : i) = \begin{cases} \text{undefined} & \text{if } q > 1 \text{ and } n - i < q - 1 \\ \bar{R}_{jn} & \text{if } q = 1 \\ \min_{i \leq t < n} \max \{ \bar{R}_{it}, M(q - 1 : t + 1) \} & \text{otherwise} \end{cases} \tag{40}$$

Using Theorem 3 and Lemma 8, our goal is to calculate  $M(k : 0) = M_{0n}^k = R_{\max}^k(P)$ . This will be sped up by the fact (Corollary 1) that the  $M_{ij}^q$  are monotonic.

From Lemma 10 and Corollary 1

$$M_{in}^q = \min_{i \leq t < n} \max \{ \bar{R}_{it}, M_{(t+1)n}^{q-1} \} = \min \{ \bar{R}_{it_q[i,n]}, M_{t_q[i,n]n}^{q-1} \} \tag{41}$$

where

$$t_q[i, n] = \min \{ t : i \leq t < n \text{ and } R_{it} \geq M_{(t+1)n}^{q-1} \} \tag{42}$$

and  $t_q[i, n]$  satisfies

$$\forall t \in [i, n], \quad \bar{R}_{it} < M_{(t+1)n}^{q-1} \quad \text{if and only if} \quad t < t_q[i, n]. \tag{43}$$

The algorithm binary-searches to find  $t_q[i, n]$  and then compares the two possibilities on the right hand side of Eq. 41 to find the correct answer.

To formalize this idea, for  $\ell_q, u_q$  satisfying  $i \leq \ell_q \leq u_q < n$ , define

$$M(q : i, \ell_q, u_q) := \min_{\ell_q \leq t \leq u_q} \max \{ \bar{R}_{it}, M_{(t+1)n}^{q-1} \}. \tag{44}$$

**Definition 14** The parameters of  $M(q : i, \ell_q, u_q)$  satisfy the **sandwich condition** if  $\ell_q \leq t_q[i, n] \leq u_q$ .

If  $M(q : i, \ell_q, u_q)$  satisfies the sandwich condition then, from Eqs. 41, 42, 43,

$$M(q : i, \ell_q, u_q) = \min_{\ell_q \leq t \leq u_q} \max \{ \bar{R}_{it}, M_{(t+1)n}^{q-1} \} = \min \{ \bar{R}_{it_q[i,n]}, M_{t_q[i,n]n}^{q-1} \} = M_{in}^q,$$

leading to

**Property 5** If  $M(q : i, \ell_q, u_q)$  satisfies the sandwich condition then

$$M(q : i, \ell_q, u_q) = M(q : i) = M_{in}^q.$$

Note that  $i \leq t_q[i, n] < n$ , so

**Property 6**  $\forall i, q, M(q : i, i, n - 1)$  satisfies the sandwich condition.

$M(q : i, \ell_q, u_q)$  : Returns  $\min_{\ell_q \leq t \leq u_q} \{ \bar{R}_{it}, M_{(t+1)n}^{q-1} \}$ .

1. If  $q = 2$  :
  - (a) If  $\ell_q = u_q = i$  : Then  $t_q[i, n] = \ell_q = i$  ; so **Return**  $(M_{in}^2 = 0)$
  - (b) If  $\ell_q = u_q \neq i$  : Then  $t_q[i, n] = \ell_q \neq i$  ;  
so **Return**  $(M_{in}^2 = \min\{ \bar{R}_{i\ell_q}, \bar{R}_{\ell_q n} \})$
  - (c) If  $\ell_q < u_q$  : Then set  $m = \lfloor (\ell_q + u_q)/2 \rfloor$   
Check if  $\bar{R}_{im} \geq \bar{R}_{(m+1)n}$   
(A) If yes, then  $\ell_q \leq t_2[i, n] \leq m$ ; so **Return**  $(M(2 : i, \ell_q, m))$   
(B) If no, then  $m < t_2[i, n] \leq u_q$ ; so **Return**  $(M(2 : i, m + 1, u_q))$
2. If  $q > 2$  :
  - (a) If  $\ell_q = u_q = i$  : Then  $\ell_q = t_q[i, n] = i$ ; so **Return**  $(M_{in}^q = 0)$
  - (b) If  $\ell_q = u_q \neq i$  : Then  $t_q[i, n] = \ell_q \neq i$  ;  
so **Return**  $(M_{in}^q = \min\{ \bar{R}_{i\ell_q}, M_{\ell_q n}^{q-1} \})$   
(A)  $M_{\ell_q n}^{q-1}$  is recursively calculated by calling  $M(q - 1 : \ell_q, \ell_q, n - 1)$
  - (c) If  $\ell_q < u_q$  : Then set  $m = \lfloor (\ell_q + u_q)/2 \rfloor$   
(A) Check if  $\bar{R}_{im} \geq M_{(m+1)n}^{q-1}$   
Do this by first calculating  $\bar{R}_{im}$  and then  $M_{(m+1)n}^{q-1}$   
by recursively calling  $M_{(m+1)n}^{q-1} = M(q - 1 : m + 1, m + 1, n - 1)$   
(B) If yes, then  $\ell_q \leq t_q[i, n] \leq m$ ; **Return**  $(M(q : i, \ell_q, m))$   
(C) If no, then  $m < t_q[i, n] \leq u_q$ ; **Return**  $(M((q : i, m + 1, u_q))$

**Fig. 6** BIN1—the first iterated binary search algorithm. Calling  $M(k : 0, 0, n - 1)$  returns the correct answer  $M_{0n}^k = R_{\max}^k(P)$

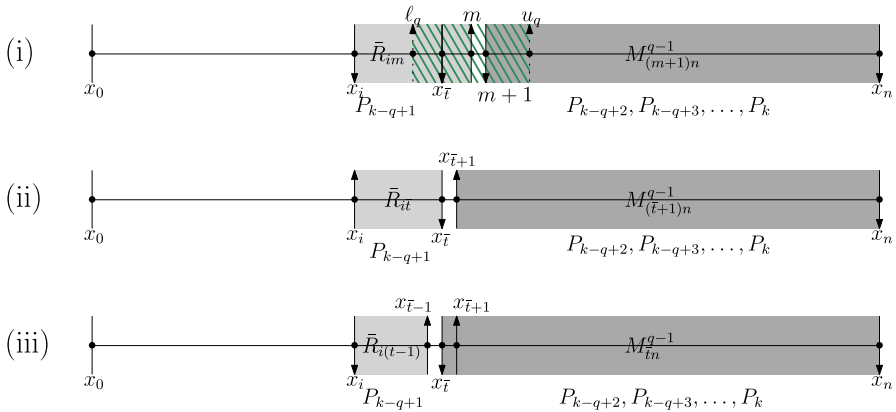
This fact will be used often below. As a special case, Property 6 implies

$$\min_{\{\hat{P}, \hat{Y}\}} R_{\max}(\{\hat{P}, \hat{Y}\}) = M_{0n}^k = M(k : 0, 0, n - 1).$$

BIN1, a recursive algorithm for evaluating  $M(q : i, \ell_q, u_q)$  satisfying the sandwich condition, is presented in Fig. 6. Calling  $M(k : 0, 0, n - 1)$  returns the final solution. Without providing details, we note that BIN1 can be easily modified to find the  $\{\hat{P}, \hat{Y}\}$  that achieves the minimum and the worst case scenario associated with it. To prove correctness of BIN1 we first show that it terminates, then that all recursive calls satisfy the sandwich condition and, finally, that if the calls satisfy the sandwich condition they terminate with the correct solution.

First note that this algorithm always terminates because all recursive calls from  $M(q : i, \ell_q, u_q)$  either decrement  $q$  or reduce  $u_q - \ell_q$ .

Next note that if the parameters of a  $M(q : i, \ell_q, u_q)$  call satisfies the sandwich condition then all of the recursive calls it makes satisfy the condition as well. For the calls on lines 2(b)(A) and 2(c)(A) this follows from Property 6. For cases 1(c) and 2(c) the sandwich condition imposes  $\bar{R}_{i\ell_q} \leq \bar{M}_{(\ell_q+1)n}^{q-1}$ . By setting  $m$  to be the midpoint between  $\ell_q$  and  $u_q$  and checking whether or not  $\bar{R}_{im} \leq \bar{M}_{(m+1)n}^{q-1}$  monotonicity implies whether  $t_q[i, n]$  is to the left or right of  $m$  and BIN1 makes a recursive call in the



**Fig. 7** The diagram illustrates Cases (c) and (b) in Algorithm BIN1. The sandwich condition guarantees that  $t_q[i, n] \in [\ell_q, u_q]$ , the crosshatched region in (i). BIN1 tests whether  $\bar{R}_{im} \geq \bar{M}_{(m+1)n}^{q-1}$  and halves the crosshatched region, recursing to the left or right appropriately. The recursion terminates when  $\bar{r} = t_q[i, n] = \ell_q = u_q$ . These are subfigures (ii) and (iii). In (ii)  $\bar{R}_{i\bar{r}} \geq M_{(\bar{r}+1)n}^{q-1}$  while in (iii)  $\bar{R}_{i\bar{r}} < M_{(\bar{r}+1)n}^{q-1}$ . BIN1 returns the minimum of  $\bar{R}_{i\bar{r}}$  and  $M_{(\bar{r}+1)n}^{q-1}$  as in case (b) in the algorithm. Case (a) of BIN1 (not pictured) is the degenerate situation  $t_q[i, n] = \ell_q = u_q = i$  which forces  $\bar{R}_{i\bar{r}} = M_{(\bar{r}+1)n}^{q-1} = 0$  and permits terminating the algorithm

appropriate half range containing  $t_q[i, n]$ , maintaining the sandwich condition. Thus the recursive calls on lines 1(c)(A), 1(c)(B), 2(c)(B) and 2(c)(C) all also satisfy the sandwich condition.

Finally, for correctness, note from the sandwich condition, (41) and (42)

- if  $\ell_q = u_q = i$ , then  $0 = \bar{R}_{ii} \geq M_{(i+1)n}^{q-1}$  so  $M_{in}^q = 0$
- if  $\ell_q = u_q \neq i$ , then  $t_q[i, n] = \ell_q$  and  $M_{in}^q = \min\{\bar{R}_{i\ell_q}, M_{\ell_q n}^{q-1}\}$ .

These are subcases (a) and (b) in cases (1) and (2). Thus the algorithm is correct when  $\ell_q = u_q$ . As noted above, when  $\ell_q < u_q$ , i.e., subcase (c) in both cases, the algorithm recurses by making a correct call that maintains the sandwich condition. The correctness of the algorithm then follows from simple induction (Fig. 7).

Now, let  $f(n)$  be the worst case time required to evaluate a single value  $R_{i,j}$  (and thus  $\bar{R}_{ij}$ ) and  $F_q(m)$  be the worst case time BIN1 requires to calculate  $M(q : i, \ell_q, u_q)$  when  $u_q - \ell_q < m$ . Working through the code yields

$$F_q(m) \leq \begin{cases} 2f(n) + O(1) & \text{if } q = 2, m = 1 \\ F_2(\lceil m/2 \rceil) + 2f(n) + O(1) & \text{if } q = 2, m > 1 \\ F_{q-1}(n) + f(n) + O(1) & \text{if } q > 2, m = 1 \\ F_q(\lceil m/2 \rceil) + F_{q-1}(n) + f(n) + O(1) & \text{if } q > 2, m > 1 \end{cases}$$



This is a very standard multi-dimensional divide-and-conquer recurrence (see, e.g., [6]) which evaluates out to  $F_q(n) = O(f(n) \log^{k-1} n)$ . Plugging in Lemma 21 immediately proves

**Theorem 5**

$$\min_{\{\hat{P}, \hat{Y}\}} R_{\max}(\{\hat{P}, \hat{Y}\}) = M_{0n}^k$$

can be calculated in  $O(n^2 k^2 \log^{k+1} n)$  time.

For fixed  $k$ , this is better than the  $O(n^3 \log n)$  algorithm from Theorem 1 and would be the best algorithm known for fixed  $k > 2$ .

**6.3 An Improved Binary Search Algorithm for  $k = 2$**

Lemma 21 permits constructing the  $R'_{ij}$  in  $O(nk^2 \log^2 n)$  time instead of the  $O(n^2 k^2 \log^2 n)$  required for  $R_{ij}$ . This suggests replacing the  $\bar{R}_{ij}$  calls in BIN1 with  $R'_{ij}$  calls. The difficulty with this approach is that Algorithm BIN1 strongly used the monotonicity of  $M_{ij}^q$  which was a consequence of the monotonicity of the  $\bar{R}_{ij}$  derived in Lemma 9. Unfortunately, as noted after the statement of Corollary 1, Lemma 9 can not be generalized to prove the monotonicity of the  $R'_{ij}$ . Consequentially, the algorithm can not just simply replace the  $R_{ij}$  with  $R'_{ij}$ .

It can, though, use the relationship between  $R_{ij}$  and  $R'_{ij}$  in Lemma 6 to simulate  $\bar{R}_{ij}$  calls in BIN1 with  $R'_{ij}$  calls.

This will be quite technical so, to provide intuition, we first work through the details for  $k = 2$ . When  $k = 2$ , BIN1 only calls case 1 (with  $q = 2$ ) and all calls have  $i \equiv 0$ . This specialized version is written as BIN2.2 in Fig. 8; for simplification, the call

- $T(\ell, u)$  : Returns  $\min_{\ell \leq t < u} \{\bar{R}_{0t}, \bar{R}_{(t+1)n}\}$ .
- %  $V(t) = \min(R'_{0t}, R'_{(t+1)n})$  and  $d'(\hat{P}(t))$  can be calculated in  $O(nk^2 \log^2 n)$  time.
- % If any call returns  $V(t) = 0$ , then terminate algorithm with  $M_{0n} = 0$ .
- (a) If  $\ell = \mathbf{u} = \mathbf{0}$  : Then  $t_2[0, n] = \ell = 0$ ; so **Return** ( $M_{0n}^2 = 0$ )
- (b) If  $\ell = \mathbf{u} \neq \mathbf{0}$  : Then  $t_2[0, n] = \ell > 0$ ;  
so **Return** ( $M_{0n}^2 = \min\{\bar{R}_{0\ell}, \bar{R}_{\ell n}\} = \min\{V(\ell), V(\ell - 1)\}$ )
- (c) If  $\ell < \mathbf{u}$  : Then set  $m = \lfloor (\ell + u/2) \rfloor$   
Check if  $\bar{R}_{0m} \geq \bar{R}_{(m+1)n}$ .  
%  $\bar{R}_{0m} \geq \bar{R}_{(m+1)n}$  if and only if  $d'(\hat{P}(t)) = 1$   
(A) If yes, then  $\ell \leq t_2[0, n] \leq m$ ; so **Return** ( $T(\ell, m)$ ).  
(B) If no, then  $m < t_2[0, n] \leq u$ ; so **Return** ( $T(m + 1, u)$ ).

**Fig. 8** BIN2.2—the second iterated binary search algorithm, specialized for the case  $k = 2$ . When  $k = 2$ ,  $q = 2$  and  $i \equiv 0$ , so the call  $M(q : i, \ell_q, u_q)$  from BIN1 can be rewritten as  $T(\ell, u)$ . Also note that  $M_{(t+1)n}^{q-1} = \bar{R}_{(t+1)n}$ . The  $\bar{R}_{ij}$  calls in the algorithm can be simulated by  $R'_{ij}$  calls as noted in the comments and explained further in the text

$M(2 : 0, \ell_q, u_q)$  is relabelled as  $T(\ell, u)$ . We now show how BIN2.2’s use of  $\bar{R}_{ij}$  terms can be replaced by  $R'_{ij}$  terms.

For given  $t$ , define the partitions

$$\hat{P}(t) := \{P_1(t), P_2(t)\} \text{ where } P_1(t) := P_{0t} \text{ and } P_2(t) := P_{(t+1)n}$$

and set

$$V(t) := \max\{R'_{0t}, R'_{(t+1)n}\}.$$

From Lemmas 2 and 6

$$V(t) = \max\{\bar{R}_{0t}, \bar{R}_{(t+1)n}\}.$$

Furthermore, if  $V(t) = 0$  for any  $t$ , then  $M_{0,n}^2 = 0$  and the algorithm can terminate.

Lemma 6 also implies that if  $V(t) > 0$ , then  $d'(\hat{P}(t)) = \bar{d}(\hat{P}(t))$  and there are only two possibilities

–  $d'(\hat{P}(t)) = \bar{d}(\hat{P}(t)) = 1$ . Then

$$R'_{0t} \geq R'_{(t+1)n} \text{ and } \bar{R}_{0t} \geq \bar{R}_{(t+1)n} \text{ and } R'_{0t} = \bar{R}_{0t}.$$

–  $d'(\hat{P}(t)) = \bar{d}(\hat{P}(t)) = 2$ . Then

$$R'_{0t} < R'_{(t+1)n} \text{ and } \bar{R}_{0t} < \bar{R}_{(t+1)n} \text{ and } R'_{(t+1)n} = \bar{R}_{(t+1)n}.$$

Now consider the 3 parts of BIN2.2

(a) If  $\ell = \mathbf{u} = \mathbf{0}$  : Then  $t_2[0, n] = \ell = 0$

In this case, exactly as in BIN1,  $0 = \bar{R}_{00} \geq \bar{R}_{1,n}$  so  $M_{0n}^2 = 0$ .

(b) If  $\ell = \mathbf{u} \neq \mathbf{0}$  : Then  $t_2[0, n] = \ell > 0$ .

As in BIN1, the algorithm needs to return  $M_{0n}^2 = \min\{\bar{R}_{0\ell}, \bar{R}_{\ell n}\}$ .

By the definition of  $t_2[0, n]$

- $d(\hat{P}(\ell)) = 1$ , so  $V(\ell) = \bar{R}_{0\ell} = R'_{0\ell}$ ,
- $d(\hat{P}(\ell - 1)) = 2$ , so  $V(\ell - 1) = \bar{R}_{\ell n} = R'_{\ell n}$ .

Thus BIN2.2 correctly returns the value

$$\min\{\bar{R}_{0\ell}, \bar{R}_{\ell n}\} = \min\{V(\ell), V(\ell - 1)\} = \min\{R'_{0\ell}, R'_{\ell n}\}.$$

(c) If  $\ell < \mathbf{u}$  : Then set  $m = \lfloor (\ell + u/2) \rfloor$ . Check if  $\bar{R}_{0m} \geq \bar{R}_{(m+1)n}$ .

Start by calculating  $V(m) = \max\{R'_{0t}, R'_{(t+1)n}\}$  and  $d'(\hat{P}(t))$ . From the observations above, if  $V(m) = 0$ , then  $M^2_{0,n} = 0$  and the algorithm can terminate. Otherwise  $V(m) \neq 0$  and

$$\bar{R}_{0m} \geq \bar{R}_{(m+1)n} \text{ iff } d'(\hat{P}(m)) = 1 \text{ iff } R'_{0m} \geq R'_{(m+1)n}$$

Thus (c) can be implemented properly by just checking whether or not  $R'_{0m} \geq R'_{(m+1)n}$ .

Notice that all evaluations of  $R_{ij}$  are now simulated by evaluations of  $R'_{ij}$ . Since Lemma 21 permits constructing the pair  $R'_{0t}, R'_{(t+1)n}$  in  $O(nk^2 \log^2 n)$  time this proves

**Theorem 6** *If  $k = 2$  then*

$$\min_{\{\hat{P}, \hat{Y}\}} R_{\max}(\{\hat{P}, \hat{Y}\}) = M^2_{0n}$$

*can be calculated in  $O(n \log^3 n)$  time.*

### 6.4 An Improved Binary Search Algorithm for General $k$

Generalizing the ideas above for  $k > 2$  is more complicated. It will need to utilize information that BIN1 ignored. More specifically,  $M(q : i, \ell_q, u_q)$  was evaluated at the end of a recursive chain of calls that actually defined a splitting of  $[x_0, x_{i-1}]$  into  $k - q$  partitions. The new algorithm will recall and use those partitions. This will be done by passing a list  $\mathcal{L}$  of the right boundaries of the partitions to the called procedure.

Also, BIN1 evaluated  $\bar{R}_{ij}$  values exactly when they were needed. The new algorithm will defer evaluating  $R'_{ij}$  values until a full  $\hat{P}$  with  $k$  partitions has been formed and will then evaluate all of the  $R'_{ij}$  values for that  $\hat{P}$  simultaneously.

**Definition 15** For  $v < k$  and a sequence  $0 \leq r_1 < r_2 < r_3 < \dots < r_v < n$  set (See Fig. 9)

$$\mathcal{L} := \langle r_1, r_2, \dots, r_v \rangle .$$

Such a list will be called a *partition sequence* with *size*  $|\mathcal{L}| := v$ . If  $\mathcal{L} = \emptyset$ , set  $v := 0$ .

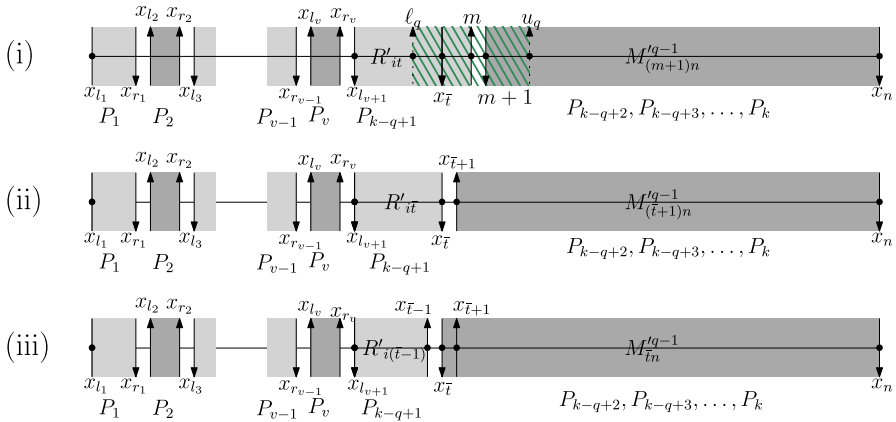
Given a partition sequence  $\mathcal{L}$  of size  $v$ ,

- Set  $l_1 := 0$  and, for  $1 \leq j \leq v$ , set  $l_{j+1} := r_j + 1$ .
- If  $v = 0$ , set  $V(\mathcal{L}) := 0$  and leave  $d(\mathcal{L})$  undefined. Otherwise, set

$$V(\mathcal{L}) := \max_{1 \leq j \leq v} \bar{R}_{l_j r_j}, \quad d(\mathcal{L}) := \arg \max_{1 \leq j \leq v} \bar{R}_{l_j r_j} .$$

- If  $v = k - 1$ , set  $r_n := n$  and  $\forall j, P_j := [x_{l_j}, x_{r_j}]$ . Define  $\hat{P}(\mathcal{L})$  as

$$\hat{P}(\mathcal{L}) := \{P_1, P_2, \dots, P_k\} .$$



**Fig. 9** In the diagram above  $\bar{t} = t_q[l_i, n]$ . This diagram illustrates the recursive calls in Case 2(b) (subfigures (ii) and (iii)) and Case 2(c) (subfigure (i)) in the revised algorithm that calculates  $T(v : \mathcal{L}, \ell_q, u_q)$ .  $\mathcal{L} = \langle r_1, r_2, \dots, r_v \rangle$ ,  $q = k - v$  and  $l_{i+1} = r_i + 1$ . Note that  $\mathcal{L}$  defines the partitions to the left of  $x_{r_v}$ . The algorithm works almost the same as BIN1 in trying to find the minimum value of  $M_{l_{v+1}n}^q$ . The main difference between this algorithm and BIN1 is that if this algorithm ever finds that  $V(\mathcal{L}) = \max_{0 \leq j \leq v} \bar{R}_{l_j, r_j} > M_{l_{v+1}n}^q$  the procedure returns immediately

– If  $v < k - 1$  and  $r_v < j < n$  define the  $j$ -extension of  $\mathcal{L}$  as

$$\mathcal{L} \oplus j := \langle r_1, r_2, \dots, r_v, j \rangle .$$

Evaluations of  $\bar{R}_{l_i r_i}$  will be deferred until  $v = k - 1$  when  $\mathcal{L}$  will correspond to a full partition. At this point, the evaluation of *all* of the  $\bar{R}_{l_i r_i}$  in  $\hat{P}(\mathcal{L})$  will be replaced by evaluation of the  $R'_{ij}$  in  $\hat{P}(\mathcal{L})$  using the following:

**Lemma 23** For partition sequence  $\mathcal{L} = \langle r_1, r_2, \dots, r_{k-1} \rangle$ , let  $Test(\mathcal{L})$  be the procedure that returns the pair  $(d, V)$  where

$$V := \max_{1 \leq j \leq k} \bar{R}_{l_j r_j} \text{ and } d := \min \{ j : \bar{R}_{l_j r_j} = V \} = \bar{d}(\hat{P}(\mathcal{L})) .$$

Then

1.  $Test(\mathcal{L})$  can be implemented in  $O(nk^2 \log^2 n)$  time.
2. If  $V = 0$ , then  $R_{\max}^k(P) = 0$ .
3. If  $V > 0$ , then  $\forall j < d, \bar{R}_{l_j r_j} < \bar{R}_{l_d r_d}$  and  $\forall j > d, \bar{R}_{l_j r_j} \leq \bar{R}_{l_d r_d}$ .

**Proof** From Lemma 21 a fixed  $R'_{l_j r_j}$  can be constructed in  $O(|r_j - l_j + 1|k^2 \log^2 n)$  time. Since  $\sum_{j=1}^k |r_j - l_j + 1| = O(n)$ , all of the  $R'_{l_j r_j}$  in  $\hat{P}(\mathcal{L})$  can be constructed in  $O(nk^2 \log^2 n)$  total time. The remainder of the lemma follows directly from Lemmas 6 and 7 and in particular the facts that

(i)

$$V(\mathcal{L}) = \max_{1 \leq j \leq k} \bar{R}_{l_j r_j} = \max_{1 \leq j \leq k} R'_{l_j r_j}, \tag{45}$$

- (ii) if  $V = 0$  then  $R_{\max}^k(P) = 0$ , and
- (iii) if  $V \neq 0$  then  $d'(\hat{P}(\mathcal{L})) = \bar{d}(\hat{P}(\mathcal{L}))$ . □

We can now describe the algorithm. For  $\mathcal{L} = \langle r_1, r_2, \dots, r_v \rangle$  set  $q = k - v$  and define

$$T(v : \mathcal{L}, \ell_q, u_q) := (d, V)$$

such that

$$\begin{aligned} V &:= \max \left\{ V(\mathcal{L}), \min_{\ell_q \leq t \leq u_q} \max \left\{ \bar{R}_{l_{v+1}t}, M_{(t+1)n}^{q-1} \right\} \right\}, \\ d &:= \begin{cases} \min \{ j : 1 \leq j \leq v \text{ such that } \bar{R}_{l_j r_j} = V(\mathcal{L}) \} & \text{if } V = V(\mathcal{L}), \\ v + 1 & \text{if } V > V(\mathcal{L}). \end{cases} \end{aligned} \tag{46}$$

Intuitively,  $\mathcal{L}$  fixes the first (leftmost)  $v$  partitions. The maximum is taken over those  $v$  partitions and the best case for the remaining  $q = k - v$  partitions on the right. If the maximum is one of the first  $v$  partitions,  $d$  will record its location. Otherwise,  $d = v + 1$  denotes that the maximum is not one of them.

We now modify algorithm BIN1 from Fig. 6 so that it calculates  $T(v : \mathcal{L}, \ell_q, u_q)$  instead of  $M(q : i, \ell_q, u_q)$ . As defined,  $M(q : i, \ell_q, u_q)$  didn't originally store information as to the partition boundaries to the left of location  $x_i$ .  $T(v : \mathcal{L}, \ell_q, u_q)$  will use that missing information, encoded in  $\mathcal{L}$ . Similar to the previous section we define

**Definition 16** The parameters of  $T(v : \mathcal{L}, \ell_q, u_q)$  satisfy the **sandwich condition** if  $\ell_q \leq t_q[l_{v+1}, n] \leq u_q$ .

If they satisfy the sandwich condition then

$$\min_{\ell_q \leq t \leq u_q} \max \left\{ R_{l_{v+1}t}, M_{(t+1)n}^{q-1} \right\} = \min \left\{ R_{l_{v+1}t_q[l_{v+1}, n]}, M_{t_q[l_{v+1}, n]}^{q-1} \right\} = M_{l_{r+1}n}^q.$$

Thus

**Property 7** If the parameters of  $T(v : \mathcal{L}, \ell_q, u_q)$  satisfy the **sandwich condition** then  $(d, V) = T(v : \mathcal{L}, \ell_q, u_q)$  satisfies

$$V = \max \left\{ V(\mathcal{L}), M_{l_{v+1}n}^q \right\}. \tag{47}$$

Furthermore, if  $V = V(\mathcal{L})$  then  $d = d(\mathcal{L})$ .

Note that since  $j + 1 \leq t_q[j + 1, n] < n$ , it follows that

**Property 8** A call of the form  $T(v : \mathcal{L} \oplus j, j + 1, n - 1)$  satisfies the sandwich condition.

As a special case,  $T(0 : \emptyset, 0, n - 1)$  satisfies the sandwich condition so  $(d, V) = T(0 : \emptyset, 0, n - 1)$  yields  $\max \{V(\emptyset), M_{l_1 n}^q\} = M_{0n}^q$ , which is what is required.

The final main observation is that if  $T(v : \mathcal{L}, \ell_q, u_q)$  is called with  $v = |\mathcal{L}| = k - 2$  then  $\mathcal{L} \oplus j$  will define a full partition so  $Test(\mathcal{L} \oplus j)$  is well-defined.

We now work through the new algorithm, BIN2. We will always assume that no call  $(d, V) = Test(\mathcal{L})$  will ever return  $V = 0$  because, if it does, Lemma 7 permits terminating BIN2 and reporting that  $M_{0,n}^k = 0$ .

BIN2 follows. We stress that the logic of BIN2 is exactly the same as that of BIN1 and therefore do not explicitly prove correctness. Instead we describe how the  $\bar{R}_{i,j}$  evaluations in BIN1 are correctly simulated by the  $Test(* : *, *, *)$  calls in BIN2. In particular, boxes labelled “Call” are calls to either  $Test$  or recursive calls made by the procedure; boxes labelled  $Return(d, v)$  are what the procedure returns (based on the calls it made).

(1)  $\mathbf{v} = \mathbf{k} - 2$  which implies  $\mathbf{q} = \mathbf{k} - \mathbf{v} = 2$  : Recall  $l_{k-1} = r_{k-2} + 1$

(a) If  $\ell_2 = \mathbf{u}_2 = \mathbf{l}_{k-1}$  the same analysis as in Sect. 6.2 shows that  $t_2[l_{k-1}, n] = \ell_2$  so  $M_{l_{k-1}n}^2 = 0$ .

Let  $(d, V) := Test(\mathcal{L} \oplus \ell_2)$ . If  $V \neq 0$  then from Property 7,  $V = V(\mathcal{L})$  and  $d = d(\mathcal{L})$ . Thus the following returns the correct answer:

**Return**  $(d, V) := Test(\mathcal{L} \oplus \ell_2)$ .

(b) If  $\ell_2 = \mathbf{u}_2 \neq \mathbf{l}_{k-1}$  : From the sandwich condition,  $\ell_2 = t_2[l_{k-1}, n]$ . To simplify the analysis set

$$A := \bar{R}_{l_{k-1}\ell_2}, \quad B := \bar{R}_{(\ell_2+1)n}, \quad A' := \bar{R}_{l_{k-1}(\ell_2-1)}, \quad B' := \bar{R}_{\ell_2 n},$$

From the definition of  $t_2[l_{k-1}, n]$ ,  $A \geq B$ ,  $A' < B'$ , and

$$M_{l_{k-1}n}^2 = \min \{ \max\{A, B\}, \max\{A', B'\} \} = \min\{A, B'\}.$$

Now

**Call**

$(d_A, V_A) := Test(\mathcal{L} \oplus \ell_2);$   
 $(d_B, V_B) := Test(\mathcal{L} \oplus (\ell_2 - 1));$

Recall that

$$V(\mathcal{L}) = \max_{1 \leq j \leq k-2} \bar{R}_{l_j r_j}.$$

From the facts above and Property 7

$$\begin{aligned}
 V_A &= \max \{V(\mathcal{L}), A, B\} = \max \{V(\mathcal{L}), A\}, \\
 V_B &= \max \{V(\mathcal{L}), A', B'\} = \max \{V(\mathcal{L}), B'\}, \\
 V &= \max \left\{ V(\mathcal{L}), M_{l_{k-1}n}^2 \right\} = \max \{V(\mathcal{L}), \min\{A, B'\}\}.
 \end{aligned}$$

The following flow directly from the definitions

$$\begin{array}{l|l}
 A \geq B & \text{so } d_A \neq k, \\
 \text{If } d_A \leq k - 2 \Rightarrow V_A = V(\mathcal{L}) \geq A, & \left| \begin{array}{l} A' < B' \quad \text{so } d_B \neq k - 1, \\ \text{If } d_B \leq k - 2 \Rightarrow V_B = V(\mathcal{L}) \geq B', \\ \text{If } d_B = k \Rightarrow V_B = B' > V(\mathcal{L}). \end{array} \right. \\
 \text{If } d_A = k - 1 \Rightarrow V_A = A > V(\mathcal{L}), &
 \end{array}$$

This provides enough information to calculate  $(d, V) = T(k - 2 : \mathcal{L}, \ell_2, \ell_2)$  from the results of the *Test()* procedures. There are four possibilities.

<b>Return (d, V)</b>			
$d_A \leq k - 2$ and	$d_B \leq k - 2$	$\Rightarrow$	$d := d_A = d_B$ and $V := V_A = V_B = V(\mathcal{L})$
$d_A = k - 1$ and	$d_B \leq k - 2$	$\Rightarrow$	$d := d_B$ and $V := V_B = V(\mathcal{L})$
$d_A \leq k - 2$ and	$d_B = k$	$\Rightarrow$	$d := d_A$ and $V := V_A = V(\mathcal{L})$
$d_A = k - 1$ and	$d_B = k$	$\Rightarrow$	$d := k - 1$ and $V := \min\{V_A, V_B\}$

(c)  $\ell_2 < u_2$  :

Note that by the sandwich condition  $\ell_2 \leq t_2[l_{k-1}, n] \leq u_2$ , so Eq. 47 implies that  $V = \max \left\{ V(\mathcal{L}), M_{l_{k-1}n}^2 \right\}$ . Set  $m := \lfloor (\ell_2 + u_2)/2 \rfloor$  and run

**Call**  $(d', V') := \text{Test}(\mathcal{L} \oplus m)$ .

By definition,  $V' = \max\{V(\mathcal{L}), A, B\}$  where

$$A := \bar{R}_{l_{k-1}m} \quad \text{and} \quad B := \bar{R}_{(m+1)n}.$$

Furthermore,

$$M_{l_{k-1}n}^2 = \min_{l_{k-1} \leq t < n} \max \{ \bar{R}_{l_{k-1}t}, \bar{R}_{(t+1)n} \} \leq \max\{A, B\}.$$

Note that if  $d' \leq k - 2$  then

$$V(\mathcal{L}) = V' \geq \max\{A, B\} \geq M_{l_{k-1}n}^2$$

and thus  $(d, V) = (d', V')$ .

If  $d' > k - 2$  the result of the *Test()* procedure can not calculate  $V$  from  $V'$ . Similar to BIN1, though, it can halve the possible range of  $t_2[l_{k-1}, n]$ . More specifically,

- If  $d' = k - 1$ 
  - $\bar{R}_{l_{k-1}m} = A = V' \geq B = \bar{R}_{(m+1)n}$
  - $\Rightarrow \ell_2 \leq t_2[\ell_{k-1}, n] \leq m$
  - $\Rightarrow (d, V) = T(k - 2 : \mathcal{L}, \ell_2, m)$ .
- If  $d' = k$ 
  - $\bar{R}_{(m+1)n} = B = V' > A = \bar{R}_{l_{k-1}m}$
  - $\Rightarrow m + 1 \leq t_2[\ell_{k-1}, n] \leq u_2$
  - $\Rightarrow (d, V) = T(k - 2 : \mathcal{L}, m + 1, u_2)$ .

This provides enough information to calculate  $(d, V) := T(k - 2 : \mathcal{L}, \ell_2, u_2)$  from the results of the *Test()* procedure. There are three possibilities.

<b>Return</b> $(d, V)$			
$d' \leq k - 2$	$\Rightarrow$	$(d, V) :=$	$(d', V')$
$d' = k - 1$	$\Rightarrow$	$(d, V) :=$	$T(k - 2 : \mathcal{L}, \ell_2, m)$
$d' = k$	$\Rightarrow$	$(d, V) :=$	$T(k - 2 : \mathcal{L}, m + 1, u_2)$

(2)  $v < k - 2$  : which implies  $q = k - v$  : Recall  $l_{v+1} = r_v + 1$

(a) If  $\ell_q = \mathbf{u}_q = \mathbf{l}_{v+1}$  the same analysis as in Sect. 6.2 shows that  $\ell_q = t_q[l_{v+1}, n]$  so

$$0 \geq \bar{R}_{l_{v+1}, l_{v+1}} \geq M_{(l_{v+1}+1)n}^{q-1} \geq 0 \tag{48}$$

which immediately implies that  $M_{(l_{v+1}+1)n}^q = 0$  and so, from Property 7,

<b>Return</b> $(d, V) := T(v + 1 : \mathcal{L} \oplus l_{v+1}, l_{v+1} + 1, n)$ .
---

(b) If  $\ell_q = \mathbf{u}_q \neq \mathbf{l}_{v+1}$  : From the sandwich condition,  $\ell_q = t_q[l_{v+1}, n]$ . Set

$$A := \bar{R}_{l_{v+1}\ell_q}, B := M_{(\ell_q+1)n}^{q-1}, A' := \bar{R}_{l_{v+1}(\ell_q-1)}, B' := M_{\ell_q n}^{q-1},$$

Again from the definition of  $t_q[l_{v+1}, n]$ ,  $A \geq B$ ,  $A' < B'$ , and  $M_{l_{v+1}n}^q = \min\{A, B'\}$ . Now

<b>Call</b>	
$(d_A, V_A) := T(v + 1 : \mathcal{L} \oplus \ell_q, \ell_q + 1, n)$	(Figure 9(ii))
$(d_B, V_B) := T(v + 1 : \mathcal{L} \oplus \ell_q - 1, \ell_q, n)$	(Figure 9(iii))

Similar to the  $v = k + 2$  case, from the facts above and the definition of  $T()$ ,

$$V_A = \max \{V(\mathcal{L}), A, B\} = \max \{V(\mathcal{L}), A\}$$

$$V_B = \max \{V(\mathcal{L}), A', B'\} = \max \{V(\mathcal{L}), B'\}$$



The pertinent facts are

$$\begin{array}{ll}
 A \geq B & \text{so } d_A \neq v + 2, \\
 \text{if } d_A \leq v & \Rightarrow V_A = V(\mathcal{L}) \geq A \\
 \text{if } d_A = v + 1 & \Rightarrow V_A = A > V(\mathcal{L}),
 \end{array}
 \qquad
 \begin{array}{ll}
 A' < B' & \text{so } d_B \neq v + 1, \\
 \text{if } d_B \leq v & \Rightarrow V_B = V(\mathcal{L}) \geq B', \\
 \text{if } d_B = v + 2 & \Rightarrow V_B = B' > V(\mathcal{L}).
 \end{array}$$

This provides enough information to calculate  $(d, V) = T(v : \mathcal{L}, \ell_q, \ell_q)$  from the results of the two recursive calls. There are four possibilities.

<b>Return (d, V)</b>			
$d_A \leq v$ and	$d_B \leq v$	$\Rightarrow d :=$	$d_A = d_B$ and $V := V_A = V_B = V(\mathcal{L})$
$d_A = v + 1$ and	$d_B \leq v$	$\Rightarrow d :=$	$d_B$ and $V := V_B = V(\mathcal{L})$
$d_A \leq v$ and	$d_B = v + 2$	$\Rightarrow d :=$	$d_A$ and $V := V_A = V(\mathcal{L})$
$d_A = v + 1$ and	$d_B = v + 2$	$\Rightarrow d :=$	$v + 1$ and $V := \min\{V_A, V_B\}$

(c)  $\ell_q < \mathbf{u}_q$  : From property 7,  $V = \max \{ V(\mathcal{L}), M_{l_{r+1}n}^q \}$ .  
 Now set  $m := \lfloor (\ell_q + u_q)/2 \rfloor$  and run

**Call**  
 $(d', V') := T(v + 1 : \mathcal{L} \oplus m, m + 1, n)$ . (Figure 9(i))

By definition,  $V' = \max\{V(\mathcal{L}), A, B\}$  where

$$A := \bar{R}_{l_{v+1}m} \quad \text{and} \quad B := M_{(m+1)n}^{q-1}.$$

Furthermore,

$$M_{l_{v+1}n}^q = \min_{l_{v+1} \leq t \leq n} \max \{ \bar{R}_{l_{v+1}t}, M_{(t+1)n}^{q-1} \} \leq \max\{A, B\}.$$

Now note that if  $d' \leq v$  then

$$V(\mathcal{L}) = V' \geq \max\{A, B\} \geq M_{l_{v+1}n}^q$$

and thus  $(d, V) = (d', V')$ .

If  $d' > v$  we can not know  $V$  from  $V'$ , but again, can halve the possible range of  $t_q[l_{v+1}, n]$ . More specifically

- If  $d = v + 1$ 
  - $\bar{R}_{l_{v+1}m} = A = V' \geq B = M_{(m+1)n}^{q-1}$
  - $\Rightarrow \ell_q \leq t_q[l_{v+1}, n] \leq u_q$
  - $\Rightarrow (d, V) = T(v : \mathcal{L}, \ell_q, m)$ .
- If  $d = v + 2$ 
  - $M_{(m+1)n}^{q-1} = B = V' > A = \bar{R}_{l_{v+1}m}$

- $\Rightarrow m + 1 < t_q[\ell_{v+1}, n] < n$
- $\Rightarrow (d, V) = T(v : \mathcal{L}, m + 1, u_q)$ .

This provides enough information to calculate  $(d, V) = T(v : \mathcal{L}, \ell_q, \ell_q)$  from the results of the recursive call. There are three possibilities.

<b>Return (d, V)</b>			
$d' \leq v$	$\Rightarrow$	$(d, V) :=$	$(d', V')$
$d' = v + 1$	$\Rightarrow$	$(d, V) :=$	$T(v : \mathcal{L}, \ell_q, m)$
$d' = v + 2$	$\Rightarrow$	$(d, V) :=$	$T(v : \mathcal{L}, m + 1, u_q)$

By construction, when the algorithm terminates, it returns the correct answer. The running time analysis is very similar to that of BIN1.

Let  $g(n)$  denote the time to run one  $Test()$  procedure. From Lemma 23,  $g(n) = O(nk^2 \log^2 n)$ . Now set  $G_q(m)$  be the worst case time required to calculate  $T(v : \mathcal{L}, \ell_q, u_q)$  when  $u_q - \ell_q < m$  and  $v = k - q$ . Working through the code gives

$$G_q(m) \leq \begin{cases} 2g(n) & \text{if } q = 2, m = 1 \\ G_2(\lceil m/2 \rceil) + g(n) & \text{if } q = 2, m > 1 \\ 2G_{q-1}(n) & \text{if } q > 2, m = 1 \\ G_q(\lceil m/2 \rceil) + 2G_{q-1}(n) & \text{if } q > 2, m > 1 \end{cases}$$

which evaluates out to  $G_q(n) = O(g(n) \log^{k-1} n) = O(nk^2 \log^{k+1} n)$ .

We have therefore just proven

**Theorem 7** *For any  $k$ , the minmax regret  $k$ -sink evacuation time*

$$\min_{\{\hat{P}, \hat{Y}\}} R_{\max}(\{\hat{P}, \hat{Y}\}) = M_{0n}^k$$

*can be calculated in  $O(nk^2 \log^{k+1} n)$  time.*

## 7 Conclusion and Further Directions

In this paper we derived new combinatorial properties that permitted efficiently solving the minmax regret  $k$ -sink location problem for dynamic flows on a path with uniform edge capacities. This provided an improved algorithm for the  $k = 2$  case and the first polynomial time algorithms for the  $k > 2$  case.

As noted earlier, our analysis assumed that sinks could be placed anywhere on an edge. An alternative model would restrict sink placement to vertices. We note that our results, algorithms and running times will all still hold in the vertex-on-sink model. To prove this, though, would require revisiting all the proofs and restricting all minimums/maximums over ranges to minimums/maximums over *vertices* in those ranges. This is mechanical; in many cases, such as the proof of Lemma 20, this would involve finding the edge in which a non-vertex minimum/maximum is found and then returning the minimum/maximum value at one of the endpoints of that edge. In

addition, the proof of Lemma 22 explicitly used the  $O(n \log n)$  algorithm from [7] to construct  $\Theta_{\text{opt}}^k(P, s)$  for fixed scenarios  $s \in S^*$ . Bhattacharya et al. [7] also assumed that sinks could be anywhere so it would also be necessary to go modify [7]’s algorithm in similar ways as well to work when sinks are restricted to be on vertices.

We conclude by discussing possible extensions. The normal (optimization and not regret) version of the sink location problem is NP-Hard to solve on general graphs even with  $k = 1$  [24], so attempting to extend the results in this paper to general graphs would not be possible. Chen and Golin [15] gives polynomial time algorithms for placing  $k$  sinks on a tree to minimize evacuation time but only the  $k = 1$  case [9] has been solved for minmax regret with an  $O(n \log n)$  algorithm. The bottleneck to solving the  $k$ -sink location minmax regret problem on trees is that while there is an understanding of the structure of the worst case scenario set for trees when  $k = 1$  there is still no good understanding of the structure when  $k > 1$ , i.e., a tree analogue for Theorem 1.

Another extension would be, on paths, to attempt to extend the results to dynamic flows with *general* capacities on the edges. More specifically, all of the prior work quoted in Sect. 1 assumes *uniform* capacity, i.e., every edge having identical capacity. A more natural problem formulation that appears in realistic dynamic flow problems allows different edges to have different capacities. Currently, though, there still is no polynomial time solution for solving the minmax regret problem on a dynamic path with general edge capacities, even when  $k = 1$ . Similar to the uniform capacity tree problem just discussed, the bottleneck in finding a solution seems to be that, for different capacity edges on a path, even when  $k = 1$ , there is no understanding of the combinatorial structure of the set of worst-case scenarios.

## Appendix

### 8 The Critical Vertex Tree Data Structure

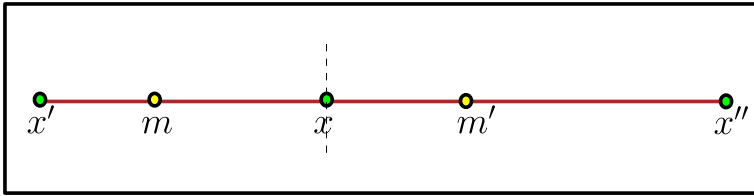
In this section we describe the CVT Data Structure of [9] and how it can be used to implement the operations described in Lemma 11 in  $O(\log n)$  time.

Let  $s$  be *any* fixed scenario. We start with [9]’s  $O(n)$  time construction (slightly modified) of an  $O(n)$  space CVT for a fixed  $s$  and afterwards describe how this can be further modified, to construct in  $O(n)$  time an extended CVT that permits queries to any  $s \in S^*$ .

Assume that  $n + 1$  is a power of 2. If not, pad path  $P$  with empty vertices on the right (with no weights) to reach a power of 2. Also before starting, perform a  $O(n)$  preprocessing step, calculating and storing all of the values  $W_i^s = \sum_{j=0}^i w_j(s)$ . After this step, for any  $a \leq b$ , the values  $W_{a,b}^s = \sum_{j=a}^b w_j(s)$  can be calculated in  $O(1)$  time. Note that, given<sup>5</sup>  $x < y$ , the values

$$W^s(x, y) = \sum_{x \leq x_j \leq y} w_j(s) = W_b^s - W_{a-1}^s$$

<sup>5</sup> The statement, “Given  $x$ ”, will always include knowing  $i$  such that  $x_i \leq x \leq x_{i+1}$ .



**Fig. 10** The left critical-vertex of path  $[x', x]$  is the node  $m$  at which the top equation in Definition 17 is maximized. Intuitively, this is the last vertex at which the first supply from  $x'$  has to stop due to congestion when evacuating to  $x$ . The diagram illustrates two paths  $[x', x]$  and  $[x, x'']$  and their respective critical vertices  $m, m'$ . The main fact upon which the CVT is based is that the critical vertex of combined path  $[x', x'']$  must be one of  $m$  and  $m'$

can then also be calculated in  $O(1)$  time.

Recall that  $\Theta_L(P, x, s)$  and  $\Theta_R(P, x, s)$  were defined to be the time needed to evacuate everything to the left (resp. right) of sink  $x$  to  $x$  under scenario  $s$ . Then, from Eq. 1,

$$\Theta_L([x', x], x, s) = \max_{x' \leq x_i < x : W^s(x', x_i) > 0} \left\{ (x - x_i)\tau + \frac{W^s(x', x_i)}{c} \right\}. \tag{49}$$

Similarly, set

$$\Theta_R([x', x], x', s) = \max_{x' < x_i \leq x : W^s(x_i, x') > 0} \left\{ (x_i - x')\tau + \frac{W^s(x_i, x')}{c} \right\}. \tag{50}$$

**Definition 17** The left/right critical vertices of  $([x', x], s)$  are

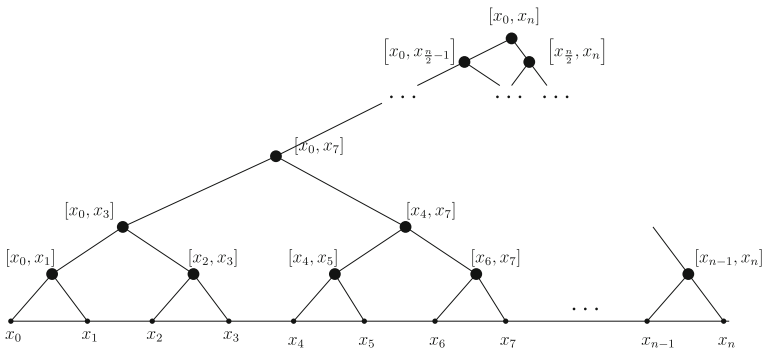
$$C_L([x', x], s) := \arg \max_{i : x' \leq x_i < x : W^s(x', x_i) > 0} \left\{ (x - x_i)\tau + \frac{W^s(x', x_i)}{c} \right\},$$

$$C_R([x', x], s) := \arg \max_{i : x' < x_i \leq x : W^s(x_i, x) > 0} \left\{ (x_i - x)\tau + \frac{W^s(x_i, x)}{c} \right\}.$$

Intuitively, the critical vertex is the last location at which items starting on the leftmost (rightmost) vertex encounter congestion. Note that if the critical vertices of  $[x', x]$  are known, then  $\Theta_L([x', x], x, s)$  and  $\Theta_R([x', x], x', s)$  can be calculated in  $O(1)$  time. The major observation [9] upon which the CVT data structure is based is (Fig. 10) that if  $x' \leq x \leq x''$  then the left (right) critical vertex of  $[x', x'']$  is either (i) the left (right) critical vertex of  $[x', x]$  or (ii) the left (right) critical vertex of  $[x, x'']$ . Thus the critical vertices of  $([x', x''], s)$  can be found in  $O(1)$  time from the critical vertices of its two subpaths.

The data structure will require a slightly extended version of this.

**Lemma 24** Let  $i \leq j \leq k$ . The critical vertices of  $(P_{ik}, s)$  can be constructed in  $O(1)$  time from the critical vertices of  $(P_{ij}, s)$  and  $(P_{(j+1)k}, s)$ .



**Fig. 11** The Critical Vertex Tree  $T^s$  for scenario  $s$  is a balanced tree with leaf nodes corresponding to path vertices and internal tree nodes corresponding to the subpath spanning descendant leaves. Node  $u$ , corresponding to subpath  $P(u) = [x_{L(u)}, x_{R(u)}]$ , will store the locations of the left and right critical vertices of  $P(u)$  under scenario  $s$

**Proof** First note that because  $(P_{j(j+1)}, s)$  only contains two vertices, its critical vertices can be found in  $O(1)$  time.

The critical vertices of  $(P_{i(j+1)}, s)$  can be calculated in  $O(1)$  time from those of  $(P_{ij}, s)$  and  $(P_{j(j+1)}, s)$ . The critical vertices of  $(P_{ik}, s)$  can be calculated from those of  $(P_{i(j+1)}, s)$  and  $(P_{(j+1)k}, s)$  in a further  $O(1)$  time.  $\square$

A critical vertex tree  $T^s$  for  $s$  was then defined by [9] as follows.

- $T^s$  is a balanced binary tree with the vertices of  $P$  as its leaf nodes<sup>6</sup>.
- For a node  $u \in T^s$  let  $L(u)$  (resp.  $R(u)$ ) denote the index of the leftmost (rightmost) vertex on  $P$  that belongs to the subtree  $T^s(u)$  rooted at  $u$ . Node  $u \in T^s(u)$  corresponds to the subpath  $P(u) = [x_{L(u)}, x_{R(u)}]$ .
- Vertex  $u \in T^s$  will store the critical values  $C_L(P(u), s)$  and  $C_R(P(u), s)$ .

A leaf node corresponds to a single vertex path and is thus its own critical vertex. For non-leaf  $u$ , let  $u_l$  and  $u_r$  denote its left and right children. Then  $P(u_l) = [x_{L(u_l)}, x_{R(u_l)}]$  and  $P(u_r) = [x_{L(u_r)}, x_{R(u_r)}]$  where  $R(u_l) + 1 = L(u_r)$ . Lemma 24 thus permits finding the critical vertices of  $P(u)$  from the critical vertices of  $P(u_l)$  and  $P(u_r)$  in  $O(1)$  time. The entire critical vertex tree  $T^s$  can therefore be constructed in  $O(n)$  time (Fig. 11).

Because  $n + 1$  is a power of 2,  $T^s$  may be stored implicitly in an array so that any node  $u$  can be accessed directly in  $O(1)$  time. In addition, if  $u$  has height  $h$  then  $P(u)$  has the form  $[x_{v2^h}, x_{(v+1)2^h-1}]$  and every path of the form  $[x_{v2^h}, x_{(v+1)2^h-1}]$  is  $P(u)$  for some  $u$ . In particular, for any  $u$  this permits evaluating  $L(u)$  and  $R(u)$  in  $O(1)$  time. For  $u, u'$ , this in turn permits checking in  $O(1)$  time whether  $P(u) \subset P(u')$ ,  $P(u') \subset P(u)$  or  $P(u) \cap P(u') = \emptyset$  (these three are the only possibilities).

We now define

**Definition 18** Set  $s^+$  to be the scenario with  $\forall j, w_j(s) = w_j^+$  and  $s^-$  to be the scenario with  $\forall j, w_j(s) = w_j^-$ . The *Critical Vertex Tree (CVT) Data Structure* is the pair  $T^{s^+}$  and  $T^{s^-}$  along with  $W_i^{s^+}, W_i^{s^-}$  for all  $i = 0, 1, \dots, n$ .

<sup>6</sup> vertex refers to a vertex of  $P$  while node refers to a node of  $T^s$ .

For  $t_1 \leq t_2$  let  $s = s_B^*(t_1, t_2)$  and consider  $T^s$ . Note that all three trees  $T^{s^+}$ ,  $T^{s^-}$  and  $T^s$  have the same topology and only differ in the *values* stored at the nodes. Let  $u^{s^+}$ ,  $u^{s^-}$  and  $u^s$  respectively denote the same node  $u$  in the three trees. It is straightforward to see that

1. If  $R(u) < t_1$  or  $R(u) > t_2$  then the subtrees rooted at  $u^{s^-}$  and  $u^s$  are identical (including their critical vertex values). Such nodes  $u \in T^s$  are called *low* nodes.
2. If  $t_1 \leq L(u) \leq R(u) \leq t_2$  then the subtrees rooted at  $u^{s^+}$  and  $u^s$  are identical (including their critical vertex values). Such nodes  $u \in T^s$  are called *high* nodes.
3. If  $u \in T^s$  is neither low nor high it is called a *split* node. Split nodes must be one of the  $O(\log n)$  tree ancestors of  $x_{t_1}$  and/or  $x_{t_2}$ .

Given the CVT Data Structure and  $s = s_B^*(t_1, t_2)$  this permits implicitly “constructing”  $T^s$  in  $O(\log n)$  time as follows. By processing the ancestors of  $x_{t_1}$  and  $x_{t_2}$  from lowest to highest and using the preprocessed information from the CVT, the critical vertices of the split nodes  $u$  can be calculated in  $O(1)$  time per node or  $O(\log n)$  total time. Since there are at most two such split nodes  $u$ ’s of a given height, their information can be stored in an array permitting  $O(1)$  lookup.

The above gives an *implicit* construction of  $T^s$ ; the critical vertex information of any node  $u \in T^s$  can be retrieved in  $O(1)$  time from  $T^{s^+}$ ,  $T^{s^-}$  or the split node array by first determining whether it is low, high or split and then looking it up in the appropriate array. Finally, note that  $W_i^s$  values can be retrieved from the  $W_i^{s^+}$  and  $W_i^{s^-}$  values in  $O(1)$  time. Thus, given the CVT Data Structure, after the  $O(\log n)$  construction of the split-node array, we can assume the existence of the CVT for  $s$ . We write this as

**Lemma 25** *Assume the CVT Data Structure has already been built. Then for any  $t_1 \leq t_2$  and  $s = s_B^*(t_1, t_2)$ ,  $T^s$  can be built in  $O(\log n)$  time.*

**Definition 19** – A *node path* in  $P$  is a subpath of the form  $P(u) = [x_{v2^h}, x_{(v+1)2^h-1}]$  where  $u \in T^s$ .

- Let  $i \leq j$ . A *Tree Decomposition* of  $P_{ij}$  of size  $t$  is a sequence  $i - 1 = i_0 < i_1 < \dots < i_t = j$  such that for  $k = 1, \dots, t$ ,  $P(k) = [x_{i_{k-1}+1}, x_{i_k}]$  is a node path. Equivalently, it is a sequence of nodes  $u_1, u_2, \dots, u_t \in T^s$  such that  $P(u_k) = [L(u_k), R(u_k)] = [x_{i_{k-1}+1}, x_{i_k}]$ .
- A *Minimal Tree Decomposition (MTD)* of  $P_{ij}$  is a tree decomposition of minimal size.

Note that the MTD of  $P_{ij}$  is unique and, for each  $h$ , contains at most two subpaths of length  $2^h$ , and thus has size  $O(\log n)$ . It can also easily be constructed in  $O(\log n)$  time. See Fig. 12 for an example.

For a given  $P_{ij}$  and  $s = s_B^*(t_1, t_2)$  let  $MTD[i, j, s]$  denote the corresponding MTD of  $P_{ij}$  of size  $t = O(\log n)$  along with the associated three arrays  $L[1..t]$ ,  $R[1..t]$  and  $R'[1..t]$  defined as follows: for  $1 \leq k \leq t$ , set

$[x_{29}, x_{29}]$	$[x_{30}, x_{31}]$	$[x_{32}, x_{63}]$	$[x_{64}, x_{127}]$	$[x_{128}, x_{255}]$	$[x_{256}, x_{319}]$	$[x_{320}, x_{327}]$	$[x_{328}, x_{329}]$
1	2	32	64	128	64	8	1

**Fig. 12** The MTD for  $P_{ij}$  with  $i = 29$  and  $j = 329$ . The path  $[x_i, x_j]$  can be decomposed into the sequence 28, 29, 31, 63, 127, 255, 319, 327, 329 which corresponds to the 8 node paths shown above, where the size of each node path is denoted below it

$$\begin{aligned}
 L[k] &:= \Theta_L(P_{iR(u_k)}, x_{R(u_k)}, s) && \text{(left evacuation time from } x_i \text{ to } x_{R_{u_k}}), \\
 R[k] &:= \Theta_R(P_{L(u_k), x_j}, x_{L(u_k)}, s) && \text{(right evacuation time from } x_j \text{ to } x_{L_{u_k}}), \\
 R'[k] &:= \Theta_R(P_{i, R(u_k)}, x_i, s) && \text{(right evacuation time from } x_{R_{u_k}} \text{ to } x_i).
 \end{aligned}$$

Given a preconstructed CVT data structure these arrays can be filled in by first building  $T^s$  in  $O(\log n)$  time using Lemma 25 and then repeated applications of Lemma 24 utilizing the critical vertex values stored at the nodes of  $T^s$ .

We can now prove Lemma 11.

**Proof of Lemma 11** The proof first first proves (1), then (3) and (4) and then returns to prove (2). In what follows, the left and right children of node  $u$  will be denoted by  $u_l$  and  $u_r$ .  $t = O(\log n)$  will always denote the size of the current MTD.

Lemma 11 (1) : Calculating  $\Theta_L(P_{ij}, x, s)$  and  $\Theta_R(P_{ij}, x, s)$ .

To calculate  $\Theta_L(P_{ij}, x, s)$ , first binary search in  $O(\log n)$  time to find  $v$  such that  $x_{v-1} < x \leq x_v \leq x_j$ . Next build  $MTD[i, v, s]$  in  $O(t)$  time. From Lemma 15

$$\Theta_L(P_{ij}, x, s) = \Theta_L(P_{iv}, x, s) - \tau(x_v - x) = L[t] - \tau(x_v - x).$$

The calculation of  $\Theta_R(P_{ij}, x, s)$  is similar (and symmetric).

Lemma 11 (3) : Constructing  $x'' = \max\{x' \in [x_i, x_j] : \Theta_L(P', x', s) \leq \alpha\}$ .

First build  $MTD[i, j, s]$  in  $O(\log n)$  time.

If  $\Theta_L(P_{ij}, x_j, s) = L[t] \leq \alpha$  then the answer is just  $x'' = x_j$ . We therefore assume that there exists  $v \leq j$  such that

$$v = \min \{r : i \leq r < j, \text{ and } \Theta_L(P_{ij}, x_r, s) \geq \alpha\}.$$

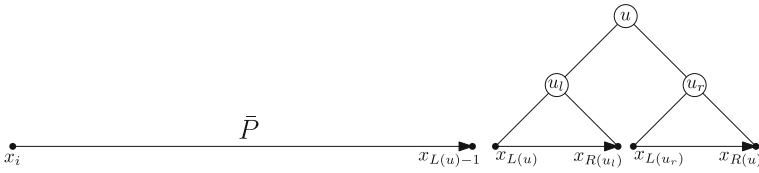
Given  $v$ , part (1) of the Lemma permits calculating  $\Theta_L(P_{iv}, x_v, s)$  in  $O(\log n)$  time. From Lemma 15, for  $x' \in (x_{v-1}, x_v]$ ,

$$\Theta_L(P_{ij}, x', s) = \Theta_L(P_{iv}, x', s) = \Theta_L(P_{iv}, x_v, s) - \tau(x_v - x').$$

Thus

$$x'' = \begin{cases} x_{v-1} & \text{if } \Theta_L(P_{iv}, x_v, s) - \tau(x_v - x_{v-1}) > \alpha, \\ \frac{1}{\tau}(\alpha - \Theta_L(P_{iv}, x_v, s) + \tau x_v) & \text{otherwise.} \end{cases}$$

and  $x''$  can therefore be found in  $O(\log n)$  time from  $v$ . It remains to show how to find  $v$  in  $O(\log n)$  time from  $MTD[i, j, s]$ .



**Fig. 13** Illustration of the proof of Lemma 11 (3), trying to find leftmost  $v$  satisfying  $\Theta_L(P_{i_v}, x_v, s) \geq \alpha$ . It is known in advance that  $x_v \in P(u)$ . Testing whether  $\Theta_L(P_{i_{R(u_l)}}, x_{R(u_l)}, s) \geq \alpha$  permits determining whether  $x_v \in u_l$  or  $x_v \in u_r$

To find  $v$ , first use  $O(\log n)$  time to find the smallest  $k$  such that  $L[k] \geq \alpha$ .

Use part (1) of the Lemma to check, in  $O(\log n)$  time, whether  $\Theta_L(P_{ij}, x_{L(u_k)}, s) \geq \alpha$ . If yes, then because  $R(u_{k-1}) = L(u_k) - 1$ ,  $v = L(u_k)$  and the process finishes.

Otherwise  $x_v \in P(u_k)$ . Set  $u := u_k$ . We now simulate a binary search for  $x_v \in P(u)$  by walking down the subtree of  $T^s$  rooted at  $u$ , on each level deciding in  $O(1)$  time whether to walk left or right. See Fig. 13.

To start, set  $\bar{P} := P_{i_{L(u)-1}}$  and from  $MTD[i, j, s]$  construct the left critical vertices of  $\bar{P}$  in  $O(\log n)$  time. We know that

$$\begin{aligned} \text{(i)} \quad & x_v \in P(u), \quad \text{(ii)} \quad \Theta_L(\bar{P}, x_{L(u)-1}, s) < \alpha, \\ \text{(iii)} \quad & \Theta_L(P_{i_{R(u)}}, x_{R(u)}, s) \geq \alpha. \end{aligned} \tag{51}$$

1. If  $P(u)$  is one node, then  $u = x_v$ . Stop.
2. Otherwise set  $\bar{P}' := P_{i_{R(u_l)}}$ , the concatenation of  $\bar{P}$  and  $P(u_l)$ . The left critical vertices of  $P(u_l)$  can be extracted from  $T^s$  in  $O(1)$  time. Lemma 24 permits combining them with the known critical vertices of  $\bar{P}$  to construct the left critical vertices of  $\bar{P}'$  in  $O(1)$  time.

This in turn permits the calculation of  $\Theta_L(\bar{P}', x_{R(u_l)}, s)$  in  $O(1)$  time

3. If  $\Theta_L(\bar{P}', x_{R(u_l)}, s) < \alpha$ , set  $\bar{P} := \bar{P}'$  and  $u = u_r$ .  
Otherwise, keep  $\bar{P}$  unchanged and set  $u := u_l$ .
4. Return to step 1

Note that  $R(u_l) = L(u_r) - 1$  so after step (3), Eq. 51 remains correct for the new value of  $v$ , regardless of whether the algorithm walks left or right.

Thus, after  $O(\log n)$  decision steps, this procedure reaches a leaf of the tree and this leaf must be  $x_v$ . Since each step uses only  $O(1)$  time and the preceding parts used only  $O(\log n)$  time, the total procedure uses  $O(\log n)$  time.

**Lemma 11 (4)** : For  $x \in [x_i, x_j]$ , calculate  $\bar{j} = \max\{j' \leq j : \Theta_R([x, x_{j'}], x, s) \leq \alpha\}$ .

First use  $O(\log n)$  time to find  $v$  such that  $x_v \leq x < x_{v+1}$ . From Lemma 15,

$$\Theta_R([x, x_{j'}], x, s) = \Theta_R(P_{vj'}, x_v, s) - \tau(x - x_v).$$

Setting  $\alpha' := \alpha + \tau(x - x_v)$ , the problem is then equivalent to finding

$$\max\{j' \leq j : \Theta_R(P_{vj'}, x_v, s) \leq \alpha'\}.$$



Now build  $MTD[v, j, s]$  in  $O(\log n)$  time. If  $R'[t] \leq \alpha$  then  $\bar{j} = j$  and the process finishes.

Otherwise, in  $O(\log n)$  time, find the largest index  $k$  such that  $R'[k] \leq \alpha'$ .

Next use Part 1 to check in  $O(\log n)$  time if  $\Theta_R(P_{v(R(u_k)+1)}, x_v, s) > \alpha'$ .

If yes, then  $\bar{j} = R(u_k)$ . Otherwise  $x_{\bar{j}} \in u_k$  and can be found in  $O(\log n)$  time using essentially the same binary search procedure as in (3).

Lemma 11 (2) : Calculating  $\Theta^1(P_{ij}, s)$  and associated  $x^*$  value.

Let

$$v := \min \left\{ r : i \leq r \leq j, \text{ and } \Theta_L(P_{ij}, x_v, s) \geq \Theta_R(P_{ij}, x_v, s) \right\}.$$

Then  $x_{v-1} \leq x^* \leq x_v$ . If  $v$  were known we could use the techniques seen previously to calculate both  $\Theta_L(P_{ij}, x_v, s)$  and  $\Theta_R(P_{ij}, x_{v-1}, s)$  in  $O(\log n)$  time and then use Lemma 15 to find  $x^*$  and  $\Theta^1(P_{ij}, s)$  in a further  $O(1)$  time.

Instead of directly finding  $v$ , we actually find the smallest  $\bar{v}$  such that  $\Theta_L(P_{ij}, x_{\bar{v}}, s) \geq \Theta_R(P_{ij}, x_{\bar{v}+1}, s)$ . Then

$$\Theta_L(P_{ij}, x_{\bar{v}+1}, s) > \Theta_L(P_{ij}, x_{\bar{v}}, s) \geq \Theta_R(P_{ij}, x_{\bar{v}+1}, s)$$

and

$$\Theta_L(P_{ij}, x_{\bar{v}-1}, s) < \Theta_R(P_{ij}, x_{\bar{v}}, s) \leq \Theta_R(P_{ij}, x_{\bar{v}-1}, s)$$

so either  $v = \bar{v}$  or  $v = \bar{v} + 1$ . We can then find the value of  $v$  in  $O(\log n)$  time using 4 applications of part (1),

To find  $\bar{v}$ , first, in  $O(\log n)$  time build  $MTD[i, j, s]$ . In  $O(\log n)$  time, find the smallest  $k$  such that  $L[k] \geq R[k + 1]$ . Let  $u = u_k$ . This implies that

$$\begin{aligned} \Theta_L(P_{ij}, x_{R(u)}, s) &\geq \Theta_R(P_{ij}, x_{R(u)+1}, s) \quad \text{and} \\ \Theta_L(P_{ij}, x_{L(u)-1}, s) &\geq \Theta_R(P_{ij}, x_{L(u)}, s) \end{aligned}$$

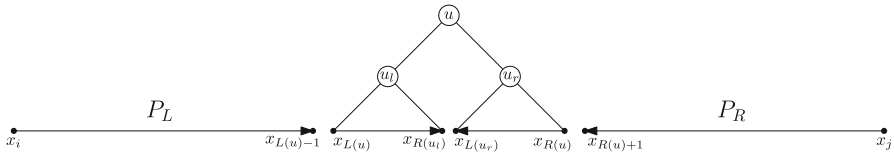
Thus  $\bar{v} \in P(u_k)$ . Set  $u := u_k$ ,  $P_L := P_{i(L(u)-1)}$  and  $P_R := P_{(R(u)+1)j}$ .  $P_L, P_R, u$  satisfy

$$(i) \bar{v} \in P(u), \quad (ii) P_L = P_{i(L(u)-1)}, \quad (iii) P_R = P_{(R(u)+1)j} \tag{52}$$

We again do a simulated binary search for  $\bar{v}$ . To initialize, in  $O(\log n)$  time, find the left critical vertex of  $P_L$  and the right critical vertex of  $P_R$ .

At each iterative step we will, in  $O(1)$  time, replace  $u$  by either  $u_l$  or  $u_r$  maintaining the correctness of all three invariants in Eq. 52. Since  $u$  starts at height  $O(\log n)$  this process only uses  $O(\log n)$  additional time (Fig. 14).

1. If  $u$  is a leaf stop and set  $\bar{v} := u$ .
2. Set  $\bar{P}_L := P_{iR(u)}$  (the concatenation of  $P_L$  and  $P(u_l)$ ) and  $\bar{P}_R := P_{L(u_r)j}$  (the concatenation of  $P(u_r)$  and  $P_R$ ).



**Fig. 14** Illustration of the proof of Lemma 11 (2), trying to find leftmost  $\bar{v}$  satisfying  $\Theta_L(P_{ij}, x_{\bar{v}}, s) \geq \Theta_R(P_{ij}, x_{\bar{v}+1}, s)$ . It is known that  $x_v \in P(u)$ . Comparing  $\Theta_L(P_{ij}, x_{R(u_l)}, s)$  and  $\Theta_L(P_{ij}, x_{L(u_r)}, s)$  permits determining whether  $x_{\bar{v}} \in P(u_l)$  or  $x_{\bar{v}} \in P(u_r)$

3. From  $MTD[i, j, s]$  find the left critical vertex of  $P(u_l)$  and right critical-vertex of  $P(u_r)$  for  $s$  in  $O(1)$  time.  
 Use the known left critical vertex of  $P_L$  and right critical vertex of  $P_R$  and Lemma 24 to construct the left critical vertex of  $\bar{P}_L$  and right critical vertex of  $\bar{P}_R$  in  $O(1)$  time.  
 Then calculate  $\Theta_L(\bar{P}_L, x_{R(u_l)}, s)$  and  $\Theta_R(\bar{P}_R, x_{L(u_r)}, s)$  in  $O(1)$  time.
4. If  $\Theta_L(\bar{P}_L, x_{R(u_l)}, s) \leq \Theta_R(\bar{P}_R, x_{L(u_r)}, s)$  set  $P_R := \bar{P}_R$  and  $u := u_l$ .  
 Otherwise set  $P_L := \bar{P}_L$  and  $u := u_r$ .
5. Go to step 1


□

## References

1. Aissi, H., Bazgan, C., Vanderpooten, D.: Min–max and min–max regret versions of combinatorial optimization problems: a survey. *Eur. J. Oper. Res.* **197**(2), 427–438 (2009)
2. Arumugam, G.P., Augustine, J., Golin, M.J., Srikanthan, P.: A polynomial time algorithm for minimax-regret evacuation on a dynamic path. [arXiv:1404.5448](https://arxiv.org/abs/1404.5448) (2014)
3. Averbakh, I., Berman, O.: Minimax regret p-center location on a network with demand uncertainty. *Location Science* **5**(4), 247–254 (1997)
4. Averbakh, I., Lebedev, V.: Interval data minmax regret network optimization problems. *Discrete Appl. Math.* **138**(3), 289–301 (2004)
5. Benkoczi, R., Bhattacharya, B., Higashikawa, Y., Kameda, T., Katoh, N.: Minsum k-sink problem on dynamic flow path networks. In: *International Workshop on Combinatorial Algorithms*, pp. 78–89. Springer (2018)
6. Bentley, J.L.: Multidimensional divide-and-conquer. *Commun. ACM* **23**(4), 214–229 (1980)
7. Bhattacharya, B., Golin, M.J., Higashikawa, Y., Kameda, T., Katoh, N.: Improved algorithms for computing k-sink on dynamic flow path networks. In: *Proceedings of WADS’2017* (2017)
8. Bhattacharya, B., Higashikawa, Y., Kameda, T., Katoh, N.: An  $o(n^2 \log^2 n)$  time algorithm for minmax regret minsum sink on path networks. In: *29th International Symposium on Algorithms and Computation (ISAAC 2018)* (2018)
9. Bhattacharya, B., Kameda, T.: Improved algorithms for computing minmax regret sinks on dynamic path and tree networks. *Theor. Comput. Sci.* **607**, 411–425 (2015)
10. Bhattacharya, B., Kameda, T., Song, Z.: A linear time algorithm for computing minmax regret 1-median on a tree network. *Algorithmica* **70**(1), 2–21 (2014)
11. Bhattacharya, B., Kameda, T., Song, Z.: Minmax regret 1-center algorithms for path/tree/unicycle/cactus networks. *Discrete Appl. Math.* **195**, 18–30 (2015)
12. Bhattacharya, Binay K., Kameda, Tsunehiko: A linear time algorithm for computing minmax regret 1-median on a tree. In: *COCOON’2012*, pp. 1–12 (2012)
13. Brodal, G.S., Georgiadis, L., Katriel, I.: An  $O(n \log n)$  version of the Averbakh–Berman algorithm for the robust median of a tree. *Oper. Res. Lett.* **36**(1), 14–18 (2008)

14. Candia-Véjar, A., Álvarez-Miranda, E., Maculan, N.: Minmax regret combinatorial optimization problems: an algorithmic perspective. *RAIRO - Oper. Res.* **45**(2), 101–129 (2011)
15. Chen, D., Golin, M.: Sink evacuation on trees with dynamic confluent flows. In: 27th International Symposium on Algorithms and Computation (ISAAC 2016), pp. 25:1–25:13 (2016)
16. Cheng, S.-W., Higashikawa, Y., Katoh, N., Ni, G., Su, B., Xu, Y.: Minimax regret 1-sink location problems in dynamic path networks. In: Proceedings of TAMC'2013, pp. 121–132 (2013)
17. Conde, E.: A note on the minmax regret centroid location on trees. *Oper. Res. Lett.* **36**(2), 271–275 (2008)
18. Ford, L.R., Fulkerson, D.R.: Constructing maximal dynamic flows from static flows. *Oper. Res.* **6**(3), 419–433 (1958)
19. Golin, M.J., Khodabande, H., Qin, B.: Non-approximability and polylogarithmic approximations of the single-sink nonsplittable and confluent dynamic flow problems. In: 28th International Symposium on Algorithms and Computation (ISAAC 2017), pp. 41:1–41:13 (2017)
20. Higashikawa, Y., Augustine, J., Cheng, S.-W., Golin, M.J., Katoh, N., Ni, G., Bing, S., Yinfeng, X.: Minimax regret 1-sink location problem in dynamic path networks. *Theor. Comput. Sci.* **588**(11), 24–36 (2015)
21. Higashikawa, Y., Golin, M.J., Katoh, N.: 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), pp. 125–137 (2014)
22. Higashikawa, Y., Golin, M.J., Katoh, N.: Multiple sink location problems in dynamic path networks. *Theoretical Computer Science* **607**, 2–15 (2015)
23. Hoppe, B., Tardos, É.: The quickest transshipment problem. *Math. Oper. Res.* **25**(1), 36–62 (2000)
24. Kamiyama, N.: Studies on quickest flow problems in dynamic networks and arborescence problems in directed graphs: a theoretical approach to evacuation planning in urban areas. Ph.D. thesis, Kyoto University (2009)
25. Kouvelis, P., Gang, Y.: *Robust Discrete Optimization and Its Applications*. Kluwer Academic Publishers, Dordrecht (1997)
26. Li, H., Yinfeng, X.: Minimax regret 1-sink location problem with accessibility in dynamic general networks. *Eur. J. Oper. Res.* **250**(2), 360–366 (2016)
27. Li, H., Xu, Y., Ni, G.: Minimax regret vertex 2-sink location problem in dynamic path networks. *J. Comb. Optim.* **31**(1), 79–94 (2016)
28. Mamada, S., Uno, T., Makino, K., Fujishige, S.: An  $O(n \log^2 n)$  algorithm for the optimal sink location problem in dynamic tree networks. *Discrete Appl. Math.* **154**(2387–2401), 251–264 (2006)
29. Megiddo, N.: Applying parallel computation algorithms in the design of serial algorithms. *J. ACM* **30**(4), 852–865 (1983)
30. Ni, G., Xu, Y., Dong, Y.: 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)
31. Osborne, M.J., Rubinstein, A.: *A Course in Game Theory*. MIT Press, Cambridge (1994)
32. Puerto, J., Rodríguez-Chia, A.M., Tamir, A.: Minimax regret single-facility ordered median location problems on networks. *INFORMS J. Comput.* **21**(1), 77–87 (2008)
33. Puerto, J., Ricca, F., Scozzari, A.: Minimax regret path location on trees. *Networks* **58**(2), 147–158 (2011)
34. Wang, H.: Minimax regret 1-facility location on uncertain path networks. In: Proceedings of the 24th International Symposium on Algorithms and Computation (ISAAC'13), pp. 733–743 (2013)
35. Wang, H.: Minimax regret 1-facility location on uncertain path networks. *Eur. J. Oper. Res.* **239**(3), 636–643 (2014)
36. Yinfeng, X., Li, H.: Minimax regret 1-sink location problem in dynamic cycle networks. *Inf. Process. Lett.* **115**(2), 163–169 (2015)
37. Ye, J.-H., Wang, B.-F.: On the minmax regret path median problem on trees. *J. Comput. Syst. Sci.* **1**, 1–12 (2015)
38. Yu, H.-I., Lin, T.-C., Wang, B.-F.: Improved algorithms for the minmax-regret 1-center and 1-median problems. *ACM Trans. Algorithms* **4**(3), 1–27 (2008)

## Affiliations

**Guru Prakash Arumugam<sup>1</sup> · John Augustine<sup>2</sup> · Mordecai J. Golin<sup>3</sup>  · Prashanth Srikanthan<sup>4</sup>**

<sup>1</sup> Present Address: Google Inc., Mountain View, USA

<sup>2</sup> IIT Madras, Chennai, India

<sup>3</sup> Hong Kong UST, Kowloon, Hong Kong

<sup>4</sup> Present Address: Microsoft Inc., Redmond, USA