# Sink Evacuation on Trees with Dynamic Confluent Flows[*][†]

**Di Chen**[1] **and Mordecai Golin**[2]

**1 Hong Kong University of Science and Technology, Hong Kong**
**2 Hong Kong University of Science and Technology, Hong Kong**

**Abstract**

Let $G = (V, E)$ be a graph modelling a building or road network in which edges have-both travel times (lengths) and *capacities* associated with them. An edge's capacity is the number of people that can enter that edge in a unit of time. In emergencies, people evacuate towards the exits. If too many people try to evacuate through the same edge, *congestion* builds up and slows down the evacuation.

Graphs with both lengths and capacities are known as *Dynamic Flow networks*. An *evacuation plan* for $G$ consists of a choice of exit locations and a partition of the people at the vertices into groups, with each group evacuating to the same exit. The *evacuation time* of a plan is the time it takes until the last person evacuates. The *$k$-sink evacuation problem* is to provide an evacuation plan with $k$ exit locations that minimizes the evacuation time. It is known that this problem is NP-Hard for general graphs but no polynomial time algorithm was previously known even for the case of $G$ a tree. This paper presents an $O(nk^2 \log^5 n)$ algorithm for the $k$-sink evacuation problem on trees, which can also be applied to a more general class of problems.

**1998 ACM Subject Classification** G.2.2 [Graph Theory] Graph Algorithms–Facility location

**Keywords and phrases** Sink Evacuation, Dynamic Flow, Facility Location, Parametric Search

**Digital Object Identifier** 10.4230/LIPIcs.ISAAC.2016.25

## 1 Introduction

*Dynamic flow networks* model movement of items on a graph.

Each vertex $v$ is assigned some initial set of supplies $w_v$. Supplies flow across edges. Each edge $e$ has a length – the time required to traverse it – and a capacity $c_e$, which limits the rate of the flow of supplies into the edge in one time unit. If all edges have the same capacity $c_e = c$ the network is said to have *uniform capacity*. As supplies move around the graph, *congestion* can occur as supplies back up at a vertex, increasing the time necessary to send a flow.

Dynamic flow networks were introduced by Ford and Fulkerson in [7] and have since been extensively used and analyzed. There are essentially two basic types of problems, with many variants of each. These are the *Max Flow over Time (MFOT)* problem of how much flow can be moved (between specified vertices) in a given time $T$ and the *Quickest Flow Problem (QFP)* of how quickly a given $W$ units of flow can be moved. Good surveys of the area and applications can be found in [19, 1, 6, 17].

[*] Extended abstract; a full version of this paper containing proofs is at `https://arxiv.org/abs/1607.08041`.
[†] Both authors partially supported by Hong Kong RGC GRF grant 16208415.

One variant of the QFP that is of interest is the transshipment problem, e.g., [12], in which the graph has several sources and sinks, with the original supplies being the sources and each sink having a specified demand. The problem is to find the minimum time required to satisfy all of the demands. [12] designed the first polynomial time algorithm for that problem, for the case of integral travel times.

Variants of QFP Dynamic flow problems can also model [10] *evacuation problems.* In these, vertex supplies are people in a building(s) and the problem is to find a routing strategy (evacuation plan) that evacuates all of them to specified sinks (exits) in minimum time. Solving this using (integral) dynamic flows, would assign each person an evacuation path with, possibly, two people at the same vertex travelling radically different paths.

A slightly modified version of the problem, the one addressed here, is for the plan to assign to each vertex $v$ exactly one exit or *evacuation edge*, i.e., a sign stating "this way out". All people starting or arriving at $v$ must evacuate through that edge. After traversing the edge they follow the evacuation edge at the arrival vertex. They continue following the unique evacuation edges until reaching a sink, where they exit. The simpler optimization problem is, given the sinks, to determine a plan minimizing the total time needed to evacuate everyone; we call this the *$k$-sink assignment problem.* A more complicated version is, given $k$, to find the (vertex) locations of the $k$ sinks/exits *and* associated evacuation plan that together minimizes the evacuation time. This is the *$k$-sink location problem*, which we also refer to as 'the' $k$-sink evacuation problem on trees.

Flows with the property that all flows entering a vertex leave along the same edge are known as *confluent*[1]; even in the static case constructing an optimal confluent flow is known to be very difficult. i.e., if P $\neq$ NP, then it is even impossible to construct a constant-factor approximate optimal confluent flow in polynomial time on a general graph [4, 5, 3, 18].

Note that if the capacities are "large enough" then no congestion can occur and every person follows the shortest path to some exit with the cost of the plan being the length of the maximum shortest path. This is exactly the $k$-center problem on graphs which is already known to be NP-Hard [9]. Unlike $k$-center, which is polynomial-time solvable for fixed $k$, Kamiyama *et al.* [13] proves by reduction from the `Partition` problem, that, even for fixed $k = 1$ finding the min-time evacuation protocol is still NP-Hard for general graphs.

The only known solvable case for general $k$ is for $G$ a path. For paths with uniform capacities, [11] gives an $O(kn)$ algorithm.[2]

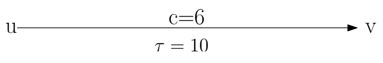
When $G$ is a tree the 1-sink location problem can be solved [16] in $O(n \log^2 n)$ time. This can be reduced [10] down to $O(n \log n)$ for the uniform capacity version, i.e., all the $c_e$ are identical. For the assignment problem, [15] gives an $O(n^2 k \log^2 n)$ algorithm for finding the minimum time evacuation protocol. i.e., a partitioning of the tree into subtrees that evacuate to each sink. The best previous known time for solving the $k$-sink location problem was $O(n(c \log n)^{k+1})$, where $c$ is some constant [14].

## 1.1 Our contributions

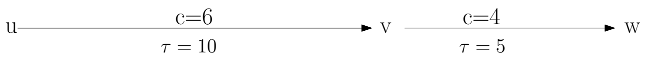
This paper gives the first polynomial time algorithm for solving the $k$-sink location problem on trees. Our result uses the $O(n \log^2 n)$ algorithm of [15], for calculating the evacuation time of a tree given the location of a sink, as an oracle.

▶ **Theorem 1.** *The $k$-sink location problem for evacuation can be solved in time $O(nk^2 \log^5 n)$.*

[1] Confluent flows occur naturally in other problems e.g. packet forwarding and railway scheduling [5].
[2] This is generalized to the general capacity path to $O(kn \log^2 n)$ in the unpublished [2].



**(a)**



**(b)**

**Figure 1** In (a), if $w_u = 20$ the last person leaving $u$ arrives at $v$ at time $t = 13$. In (b) Assume people at $u, v$ are all evacuating to $w$ and $w_u = 20$ and $w_v > 0$. The first person from $u$ arrives at $v$ at time $t = 11$. If $w_v \leq 40$ all of the people on $v$ enter $(v, w)$ *before or at* $t = 10$, so there will be no congestion when the first people from $u$ arrive at $v$ and they just sail through $v$ without stopping. The last people from $u$ reach $w$ at time $t = 20$.; but if $w_v > 40$, some people who started at $v$ will still be waiting at $v$ when the first people from $u$ arrive there. In this case, there is congestion and the people from $u$ will have to wait. After waiting, the last person from $u$ will finally arrive at $w$ at time $14 + \lfloor (20 + w_v)/4 \rfloor$.

It is instructive to compare our approach to Frederickson's [8] $O(n)$ algorithm for solving the $k$-center problem on trees, which was built from the following two ingredients.

1. An $O(n)$ time previously known algorithm for checking *feasibility*, i.e., given $\alpha > 0$, testing whether a $k$-center solution with cost $\leq \alpha$ *exists*
2. A clever *parametric search* method to filter the $O(n^2)$ pairwise distances between nodes, one of which is the optimal cost, via the feasibility test.

Section 3, is devoted to constructing a first polynomial time feasibility test for $k$-sink evacuation on trees. It starts with a simple version that makes a polynomial number of oracle calls and then is extensively refined so as to make only $O(k \log n)$ (amortized) calls.

On the other hand, there is no small set of easily defined cost values known to contain the optimal solution. We sidestep this issue by doing parametric searching *within* our feasibility testing algorithm, Section 4, which leads to Theorem 1.

As a side result, a slight modification to our algorithm allows improving, for almost all $k$, the best previously known algorithm for the $k$-sink assignment problem, from $O(n^2 k \log^2 n)$ [15] down to $O(nk^2 \log^4 n)$, as justified in the full paper.

## 2 Formal definition of the sink evacuation problem

Let $G = (V, E)$ be an undirected graph. Each edge $e = (u, v)$ has a travel time $\tau_e$; flow leaving $u$ at time $t = t_0$ arrives at $v$ at time $t = t_0 + \tau_e$. Each edge also has a *capacity* $c_e \geq 0$. This restricts at most $c_e$ units of resource to enter edge $e$ in one unit of time. For our version of the problem we restrict $c$ to be integral; the capacity can then be visualized as the *width* of the edge with only $c_e$ people being allowed to travel in parallel along the edge.

Consider $w_u$ people waiting at vertex $u$ at time $t = 0$ to travel the edge $e = (u, v)$. Only $c_e$ people enter the edge in one time unit, so the items travel in $\lceil w_u / c_e \rceil$ packets, each of size $c_e$, except possibly for the last one. The first packet enters $e$ at time $t = 0$, the second at time $t = 1$, etc.. The first packet therefore reaches $v$ at time $t = \tau_e$ time, the second at $t = \tau_e + 1$ and the last one at time $t = \tau_e + \lceil w_u / c_e \rceil - 1$. Figure 1(a) illustrates this process. In the diagram people get moved along $(u, v)$ in groups of size at most 6. If $w_u = 20$, there are 4 groups; the first one reaches $v$ at time $t = 10$, the second at time $t = 11$, the third at $t = 12$ and the last one (with only 2 people) at $t = 13$.

Now suppose that items are travelling along a path $\ldots u \to v \to w \to \ldots$ where $e_1 = (u, v)$ and $e_2 = (v, w)$. Items arriving at $v$ can't enter $e_2$ until the items already there have left. This waiting causes *congestion* which is one of the major complications involved in constructing good evacuation paths. Figure 1(b) illustrates how congestion can build up.

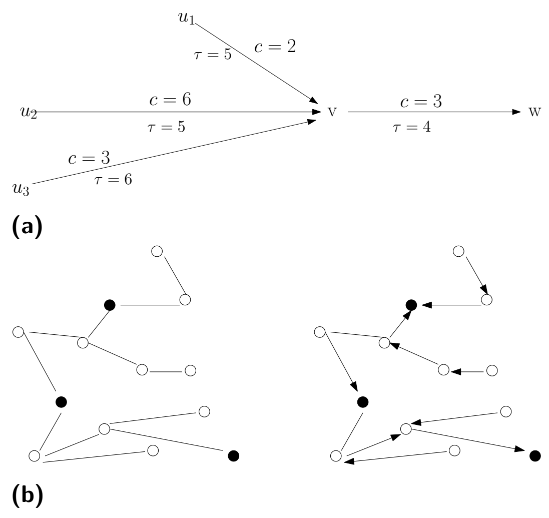


**Figure 2** (a) evacuation problem with 4 vertices and sink at $w$. If $w_v = 8$, $w_{u_1} = 4$, $w_{u_2} = 10$, $w_{u_3} = 11$ and sink at $w$, the last person arrives $w$ at time 15. In (b) the left figure is a tree with the $k = 3$ black vertices as sinks. The right figure provides an evacuation plan. Each non-sink vertex $v$ has exactly one outgoing edge and, following the directed edges from each such $v$ leads to a sink.

As another example, consider Figure 2(a) with every node evacuating to $w$. When the first people from $u_1$ arrive at $v$, some of the original people still remain there, leading to congestion. Calculation shows that the last people from $u_1$ leave $v$ at time 4 so when the first people from $u_2$ arrive at $v$ at time 5, no one is waiting at $v$. But, when the first people from $u_3$ arrive at $v$ some people from $u_2$ are waiting there, causing congestion. After that, people arrive from both $u_2$ and $u_3$ at the same time, with many having to wait. The last person finally reaches $w$ at time 15, so the evacuation protocol takes time 15.

Given a graph $G$, distinguish a subset $S \subseteq V$ with $|S| = k$ as sinks (exits). An evacuation plan provides, for each vertex $v \notin S$, the unique edge along which all people starting at or passing through $v$ evacuate. Furthermore, starting at any $v$ and following the edges will lead from $v$ to one of the $S$ (if $v \in S$, people at $v$ evacuate immediately through the exit at $v$). Figure 2(b) provides an example.

Note that the evacuation plan defines a confluent flow. The evacuation edges form a directed forest; the sink of each tree in the forest is one of the designated sinks in $S$.

Given the evacuation plan and the values $w_v$ specifying the initial number of people located at each node, one can calculate, for each vertex, the time (with congestion) it takes for all of its people to evacuate. The maximum of this over all $v$ is the minimum time required to evacuate *all* people to some exit using the rules above. Call this the *cost for $S$ associated with the evacuation plan.* The *cost* for $S$ will be the minimum cost over all evacuation plans using that set $S$ as sinks.

The *$k$-sink location* problem is to find a subset $S$ of size $k$ with minimum cost. Recall that [15] provides an $O(n \log^2 n)$ problem for solving this problem if for tree $G$ with $k = 1$. We will use this algorithm as an oracle for solving the general $k$-location problem on trees.

Given the hardness results, there may not exist an efficient algorithm for general graphs, but our algorithms can serve as fast subroutines for exhaustive search or heuristic methods.

## 2.1 General problem formulation

The input to our algorithm(s) will be a tree $T_{\text{in}} = (V_{\text{in}}, E_{\text{in}})$, and a positive integer $k$. Let $n = |V_{\text{in}}| = |E_{\text{in}}| + 1$. Our goal will be to find a subset $S \subseteq V_{\text{in}}$ with cardinality at most $k$ that can minimize the cost denoted by $F(S)$. This will essentially involve partitioning the $V_{\text{in}}$ into $\leq k$ subtrees that minimizes their individual max costs.

In the main part of the paper, we denote by $f(U, s)$, where $s \in U$, the time taken by all people from nodes in $U$ to evacuate to $s$. So given a set of sinks $S$, and a partition $\mathcal{P}$ of $V_{\text{in}}$ where $|S \cap P| = 1$ for all $P \in \mathcal{P}$, the total evacuation time is given by $\max_{P \in \mathcal{P}} f(P, S \cap P)$.

Our solution will actually work for any $f(U, v)$ that is a *monotone min-max cost* function. This is a clean abstraction of evacuation functions that allows us to cleanly formulate and understand proofs. Formulated this way, our techniques permit solving other related problems. See the full paper for details.

We also write our proofs in the full paper under such framework. Our algorithms are designed to make calls directly to an oracle $\mathcal{A}$ that, given any $U$ that induces a connected component of $T_{\text{in}}$ and any $v \in U$, computes a monotone min-max cost function $f(U, v)$. In general such a polynomial time oracle must exist for the problem to even be in NP. As we fully account for the time used to call the oracle in any way, there is no material difference whether the oracle is considered a part of the algorithm or given in the input.

## 3 Bounded cost $k$-sink (feasibility test)

To tackle the more complicated general formulation, we first consider a simplification, which is a decision problem whether all nodes can be evacuated given a time limit $\mathcal{T}$, with $k$ sinks. We call the general version of this problem "bounded cost minmax $k$-sink". We will use an algorithm solving this problem as a subroutine for solving the full problem; we measure the time complexity by the number of calls to the oracle $\mathcal{A}$, as follows.

▶ **Theorem 2.** *If $\mathcal{A}$ runs in time $t_{\mathcal{A}}(n)$, the bounded cost minmax $k$-sink problem can be solved in time $O(k \max(t_{\mathcal{A}}(n), n) \log n)$.*

▶ **Definition 3.** A *feasible configuration* is a set of sinks $S \subseteq V$ with a partition $\mathcal{P} \in \Lambda(S)$ where $F_S(\mathcal{P}) \leq \mathcal{T}$; $S$ is also separately called a *feasible sink placement*, and $\mathcal{P}$ is *a partition witnessing the feasibility of $S$*. An optimal feasible configuration is a *feasible sink placement $S^* \subseteq V$ with minimum cardinality*; we write $k^* := |S^*|$.

If $k^* > k$ then the algorithm returns 'No'. Otherwise, it returns a feasible configuration $(S_{\text{out}}, \mathcal{P}_{\text{out}})$ such that $|S_{\text{out}}| \leq k$.

▶ **Definition 4.** Suppose $U$ induces a subtree of $T_{\text{in}}$ and $S \subseteq U$. We say $U$ can be *served by* $S$ if, for some partition $\mathcal{P}$ of $U$, for each $P \in \mathcal{P}$ there exists $s \in S$ such that $f(P, s) \leq \mathcal{T}$.

▶ **Definition 5.** Let $U$ be the nodes of a connected component of $G$ and $v \in V$ (not necessarily in $U$). We say that $v$ *supports* $U$ if one of the following holds:

- If $v \in U$, then $f(U, v) \leq \mathcal{T}$.
- If $v \notin U$, let $\Pi$ be the set of nodes on the path from $v$ to $U$. Then $f(U \cup \{v\} \cup \Pi, v) \leq \mathcal{T}$.

If $U$ can be served by $S$, then any node in $U$ is supported by some $s \in S$. The converse is not necessarily true.

## 3.1 Greedy construction

Our algorithms greedily build $S_{\text{out}}$ and $\mathcal{P}_{\text{out}}$ on-the-fly, making irrevocable decisions on what should be in the output. $S_{\text{out}}$ is initialized to be empty. In each step, we add elements to $S_{\text{out}}$ but never remove them, and once $|S_{\text{out}}| > k$ we immediately terminate with a 'No'. If, at termination, $|S_{\text{out}}| \leq k$, we output $S_{\text{out}}$.

Similarly, $\mathcal{P}_{\text{out}}$ is initially empty, and the algorithm performs irrevocable updates to $\mathcal{P}_{\text{out}}$ while running. An update to $\mathcal{P}_{\text{out}}$ is a *commit.* When set $P_{\text{new}} \subseteq V_{\text{in}}$ is committed it is associated with some some sink in $S_{\text{out}}$ (which might have to be added to $S_{\text{out}}$ at the same time). If $P_{\text{new}}$ shares its sink with an existing block $P \in \mathcal{P}_{\text{out}}$, we merge $P_{\text{new}}$ into $P$. Another way to view this operation is that either a new sink is added, or unassigned nodes are assigned to an existing sink.

In essence, we avoid backtracking so that $S_{\text{out}}$ does not lose elements, and blocks added to $\mathcal{P}_{\text{out}}$ can only grow. For this to work, we must require, throughout the algorithm:

**(C1)** An optimal feasible sink placement $S^*$ exists where $S_{\text{out}} \subseteq S^*$.

**(C2)** For any $P \in \mathcal{P}_{\text{out}}$ there exists a unique $s \in S_{\text{out}}$ such that $|P \cap S_{\text{out}}| = 1$, and $f(P, s) \leq \mathcal{T}$.

Additionally, $\mathcal{P}_{\text{out}}$ will be a partition of $V_{\text{in}}$ upon termination with 'yes'. When these conditions all hold, then $|\mathcal{P}_{\text{out}}| \leq k$ and $(S_{\text{out}}, \mathcal{P}_{\text{out}})$ is feasible and output by the algorithm.

### 3.1.1 A separation argument

As the algorithm progresses, it removes nodes from the remaining graph (the *working tree*), simplifying the combinatorial structure. We will need the definitions below:

▶ **Definition 6** (Self sufficiency and $T_{-v}(u)$). A subtree $T' = (V', E')$ of $T_{\text{in}}$ is *self-sufficient* if $V'$ can be served by $S_{\text{out}} \cap V'$.

Given a tree $T = (V, E)$, consider an internal node $v \in V$ and one of its neighbors $u \in V$. Removing $v$ from $T$ leaves a forest $\mathcal{F}_{-v}$ of disjoint subtrees of $T$. Then there is a unique tree $T' = (V', E') \in \mathcal{F}_{-v}$ such that $u \in V'$, denoted by $T_{-v}(u) = (V_{-v}(u), E_{-v}(u))$. The concept of self sufficiency is introduced for subtree of this form.

Roughly speaking, if $T_{-v}(u)$ is self-sufficient, and $u$ is a sink,

there is no need to add any other sinks to $T_{-v}(u)$, also no node outside $T_{-v}(u)$ will be routed to any sink in $T_{-v}(u)$ other than $u$. This means all nodes in $V_{-v}(u)$ except $u$ can be removed from consideration; a more formal statement of this fact is given in the full version.

Throughout the algorithm, we maintain a 'working' tree $T = (V, E)$ as well as a working set of sinks $S = S_{\text{out}} \cap V$. Initially, $T = T_{\text{in}}$. As the algorithm progresses, $T$ is maintained to be a subtree of $T_{\text{in}}$ by peeling off self-sufficient subtrees, which ensures that solving the bounded problem on $T$ is equivalent to solving the bounded problem on $T_{\text{in}}$.

For this to work, we enforce that sink $s$ is added to $S_{\text{out}}$ and $S$ only when, for some neighbor $u$ of $s$, the tree induced by $V_{-s}(u) \cup \{s\}$ is self-sufficient with respective to the sink set $S \cup \{s\}$. This permits removing $V_{-s}(u)$ from $T$ after adding $s$ to $S_{\text{out}}$ and $S$. So in the algorithm we can assume that sinks exist only at the leaves of the working tree $T$.

## 3.2 Subroutine: Peaking Criterion

We now describe a convenient mechanism that allows us to greedily add sinks.

▶ **Definition 7** (Peaking criterion)**.** Given $T = (V, E)$, the ordered pair of points $(u, v) \in V \times V$ satisfies the *peaking criterion* (abbreviated *PC*) if and only if $u$ and $v$ are neighbors, $V_{-v}(u) \cap S$ is empty, and finally $f(V_{-v}(u), u) \leq \mathcal{T}$ but $f(V_{-v}(u) \cup \{v\}, v) > \mathcal{T}$.

▶ **Lemma 8.** *Let $S$ be a feasible sink placement for $T$, and let $u, v \in V$ be neighbors. If $(u, v)$ satisfies the peaking criterion, then $S' := (S \backslash V_{-v}(u)) \cup \{u\}$ is also a feasible sink placement. In particular, if $S$ is an optimal feasible sink placement, then so is $S'$.*

If $(u, v)$ satisfies the peaking criterion, we can immediately place a sink at $u$ and then commit $V_{-v}(u)$. The following demonstrates that, whenever $S = \emptyset$, at least 1 sink can be found using the peaking criterion, unless a single node can $s \in V$ support the entire graph.

▶ **Lemma 9.** *Suppose for some $v, u$, $f(V_{-v}(u) \cup \{v\}, v) > \mathcal{T}$, and $S \cap V_{-v}(u) = \emptyset$. Then there exists a pair of nodes $s, t \in V_{-v}(u) \cup \{v\}$ such that $(s, t)$ satisfies the peaking criterion.*

▶ **Corollary 10.** *Given $S = \emptyset$, either one of the following occurs:*
1. *For any $s \in V$ we have $f(V, s) \leq \mathcal{T}$, or*
2. *There exist a pair of nodes $u, v \in V$ that satisfies the peaking criterion.*

At stages where it is applicable, for each ordered pair $(u, v)$ that satisfies the peaking criterion we place a sink at $u$ and remove nodes in $V_{-v}(u)$. If instead the first case of the above corollary occurs, we can add an arbitrary $s \in V$ to $S$ and $S_{\text{out}}$ and terminate.

## 3.3 Subroutine: Reaching Criterion

Corollary 10 provides two ways to add sinks to $S_{\text{out}}$. The peaking criterion is a way to add sinks to $T$ and remove certain nodes from $T$. On the other hand, the reaching criterion (RC) is a way to *remove* sinks from $T$ and $S$, while keeping them in $S_{\text{out}}$. Roughly speaking, the reaching criterion finalizes all nodes that should be assigned to certain sinks, and then removes all these nodes from consideration. To forumlate RC, we first introduce the hub tree, which has convenient properties that arise from applying the peaking criterion.

▶ **Definition 11** (Hubs)**.** Let $L \subseteq V$ be the leaves of the rooted tree $T = (V, E)$. Let $S \subseteq L$, be a set of sinks, with no sink in $V \backslash L$. Let $H(S) \subseteq V$ be the set of lowest common ancestors of all pairs of sinks in $T$. The nodes in $H(S)$ are the *hubs* associated with $S$.

The *hub tree* $T_{H(S)} = (V_{H(S)}, E_{H(S)})$ is the subgraph of $T$ that includes all vertices and edges along all possible simple paths among nodes $H(S) \cup S$.

▶ **Definition 12** (Outstanding branches)**.** Given $T = (V, E)$ and $S$, we say that a node $w \in V$ branches out to $\eta$ if $\eta$ is a neighbor of $w$ in $T$ that does not exist in $V_{H(S)}$. The subtree $T' := T_{-w}(\eta)$ is called an *outstanding branch*; we say that $T'$ is *attached* to $w$.

▶ **Definition 13** (Bulk path)**.** Given two distinct $u, v \in V_{H(S)}$, the *bulk path* $\mathrm{BP}(u, v)$ is the union of nodes along the unique path $\Pi$ between $u, v$ (inclusive), along with all the nodes in all outstanding branches that are attached to any node in $\Pi$.

Given $T$ and $S$, we say a node $v \in E_{H(S)}$ *can evacuate* to $s \in S$ if $f(\mathrm{BP}(v, s), s) \leq \mathcal{T}$; when such $s \in S$ exists for $v$, we say that $v$ *can evacuate*. Given this we can formulate an 'opposite' to the peaking criterion, which allows us to remove nodes, including sinks, from $T$.

▶ **Definition 14** (Reaching criterion)**.** Given $T = (V, E)$ and a set of sinks $S$, placed at the leaves of $T$. Let $T$ be RC-viable with respect to $S$ and $(u, v) \in V \times V$ be an ordered pair of nodes. $u, v$ satisfy the *reaching criterion (RC)* if and only if they are neighbors in $T$, and $T_{-v}(u)$ is self-sufficient while the tree induced by $\mathrm{BP}(v, u) \cup V_{-v}(u)$ is not.

A crucial property arises after an exhaustive application of the peaking criterion.

▶ **Definition 15** (RC-viable)**.** Given $T$ and sinks $S$, we say that $T$ is *RC-viable* if:

1. all sinks $S$ occur at the leaves of $T$
2. if $T' = (V', E')$ is an outstanding branch attached to $w \in V_{H(S)}$, then $f(V' \cup \{w\}, w) \leq \mathcal{T}$

▶ **Lemma 16.** *Given $T = (V, E)$ and sinks $S$, where $S$ is a subset of leaves of $T$. Suppose no ordered pair $(u, v) \in V \times V$ satisfy the peaking criterion. Then $T$ is RC-viable.*

Now when $T$ is RC-viable w.r.t. $S$, there is no need to place sinks within outstanding branches; this is because if an outstanding branch is attached to a node $w$, then a sink at $w$ can already serve the entire outstanding branch.

▶ **Theorem 17.** *Suppose $T = (V, E)$ is RC-viable with respect to $S \subseteq V$. If $u, v \in V$ satisfies the reaching criterion, then we can remove $T_{-v}(u)$ from $T$, and also commit all blocks in the partitioning of $T_{-v}(u)$ that witnesses the self-sufficiency of $T_{-v}(u)$. By definition, $T_{-v}(u)$ includes at least one sink from $S$.*

After removing $T_{-v}(u)$ by the reaching criterion, we need to run the peaking criterion again on $T$, in order to preserve RC-viability. Then, we can apply RC again. In this way, we interleave the invocations of PC and RC to gradually eliminate nodes from the working tree.

### 3.3.1 Testing for self-sufficiency

In order to make use of the reaching criterion, we require efficient tests for self-sufficiency. Note that [15] readily gives such test albeit at a higher time complexity. In our algorithm, we perform self-sufficiency tests on a rooted subtree $T'$ only if it satisfies some special conditions, allowing us to exploit RC-viability and reuse past computations. By our arrangements, when such $T'$ passes our test we know it demonstrates a stronger form of self-sufficiency.

▶ **Definition 18** (Recursive self-sufficiency)**.** Given a rooted subtree $T' = (V', E')$ of $T$, $V' \cap S \neq \emptyset$, we say that $T'$ is *recursively self-sufficient* if for all $u \in V_{H(S)} \cap V'$, the subtree of $T'$ rooted at $u$ is self-sufficient.

A bottom-up approach can be used to test for recursive self-sufficiency, which in turn implies 'plain' self-sufficiency.

▶ **Lemma 19.** *Given a RC-viable rooted subtree $T' = (V', E')$ of $T$, $V' \cap S \neq \emptyset$, where $v$ is the root. Suppose there exists a child $u$ of $v$ in $V_{H(S)} \cap V'$ such that $T_{-v}(u)$ is recursively self-sufficient, and there is a sink $s \in S \cap V_{-v}(u)$ such that $v$ can evacuate to $s$.*

*Then $BP(v, s) \cup V_{-v}(u)$ is recursively self-sufficient. If, additionally, for every child $u'$ of $v$ in $V_{H(S)} \cap V'$, $T_{-v}(u')$ is recursively self-sufficient, then $T'$ is recursively self-sufficient.*

We say that $s$ is a witness to Lemma 19 for $T'$ and $v$; we store this witness, as well as the witness for every subtree of $T'$ rooted at some $v \in V'$. One can retrieve a partition $\mathcal{P}'$ of $T'$ that witnesses the self-sufficiency of $T'$, in $O(|V'|)$ time. For this to be useful, note that only recursive self-sufficiency will be relevant. When a RC-viable tree is self-sufficient but not recursively self-sufficient, if we process bottom-up, we can always cut off part of the tree using the reaching criterion, so that the remainder is recursively self-sufficient. This is demonstrated in the detailed algorithm, in the full paper.

## 3.4 Combining the Pieces

The main ingredients of our algorithm are the peaking and reaching criteria along with ideas to test self-sufficiency. We use the peaking criterion to add sinks to $T$, and then the reaching criterion to remove sinks and nodes from $T$, until either $T$ is empty or $T$ can be served by a single sink. In the following we describe a full algorithm that makes use of these ideas.

### 3.4.1 Simpler, iterative approach ('Tree Climbing')

Essentially, in this algorithm we iteratively check and apply the PC and RC bottom-up from the leaves. We do not specify a root here; the root can be arbitrary, and changed whenever necessary. As we go up from the leaves, for each pair $(u, v)$ that forms an edge of the tree, we would call the oracle $\mathcal{A}$ for $f(V_{-v}(u), u)$, $f(V_{-v}(u) \cup \{v\}, v)$ or $f(\mathrm{BP}(v, s), s)$ for some sink $s$, and apply either the peaking criterion or the reaching criterion. By design RC is checked whenever the tree is RC-viable, and PC is checked whenever the tree is not RC-viable, and we do not need to test both on the same pair $(u, v)$.

▶ **Lemma 20.** *The bounded-cost tree-climbing algorithm makes $O(n)$ calls to $\mathcal{A}$.*

**Proof.** We only make $O(1)$ calls to evaluate $f(\cdot, \cdot)$ for each pair $(u, v) \in V_{\text{in}}$. ◀

After seeing the iterative approach, it is easier to understand the more advanced algorithm, which uses divide-and-conquer and binary search to replace the iterative processes.

### 3.4.2 Peaking criterion by recursion

Macroscopically, we replace plain iteration with a fully recursive process. We do this once in the beginning, as well as every time we remove a sink. Overall the algorithm makes $O(k \log n)$ 'amortized' calls to the oracle. Recall that the main purpose of the peaking criterion is to place sinks and make the tree $T$ RC-viable.

#### A localized view

We start with a more intuitive, localized view of the recursion. We evaluate $f(\cdot, \cdot)$ on sets of nodes of the form $V_{-v}(u)$ or $V_{-v}(u) \cup \{v\}$. If $f(V_{-v}(u), u) \le \mathcal{T}$ then we mark all nodes in $V_{-v}(u)$; note that if $f(V_{-v}(u) \cup \{v\}, u) > \mathcal{T}$ but all nodes in $V_{-v}(u)$ are marked, then PC can be invoked and $V_{-v}(u)$ is cut off. Sometimes we also mark the node $v$, if all but one of its neighbors (that have not been cut off) are marked.

Over the course of the algorithm, we are given a node $v \in V$ (along with other information including which other nodes are marked), and for each neighbor $u$ of $v$ we decide whether to evaluate $a_u := f(V_{-v}(u) \cup \{v\}, v)$. As a basic principle to save costs, we do not wish to call the oracle if all nodes in $V_{-v}(u) \cup \{v\}$ are marked, or if $V_{-v}(u) \cup \{v\}$ contains a sink.

When we do get $a_u \le \mathcal{T}$, mark all nodes in $V_{-v}(u)$. Moreover, if no more than 1 neighbor of $v$ is unmarked, and by this time $v \notin S_{\text{out}}$, we also mark $v$. This part is the same in tree-climbing, and maintains *an important invariant* regarding the set of marked nodes: if $u$ is marked but a neighbor $v$ is not, then all nodes in $V_{-v}(u)$ are marked, and $f(V_{-v}(u) \cup \{v\}, v) \le \mathcal{T}$.

On the other hand, if in fact we find that $a_u > \mathcal{T}$, we would wish to recurse into $T_{-v}(u)$, because one sink must be placed in it. Now we return to a more global view.

### A global view

To maintain RC-viability we need to apply the oracle on various parts of $T$. In the iterative algorithm, this process is extremely repetitive. Now we wish to segregate different sets of nodes on the tree, so the oracle is only applied to separate parts.

▶ **Definition 21** (Compartments and Boundaries). Let $T' = (V', E')$ be a subtree of $T = (V, E)$. The boundary $\delta T'$ of $T'$ is the set of all nodes in $T'$ that is a neighbor of some node in $V \backslash V'$.

Now given a set of nodes $W$ of a tree $T = (V, E)$, the set of compartments $\mathcal{C}_T(W)$ is a set of subtrees of $T$, where the union of all nodes is $V$, and for each $T' = (V', E') \in \mathcal{C}_T(W)$, $V'$ is a maximal set of nodes that induces a subtree $T''$ of $T$ such that $\delta T'' \subseteq W$.

Intuitively, the set of compartments is induced by first removing $W$, so that $T$ is broken up into a forest of smaller trees, and for each of the small trees we re-add nodes in $W$ that were attached to it, where the reattached nodes are called the boundary. As opposed to partitioning, two compartments may share nodes at their boundaries.

In the algorithm, we generate a sequence of sets $W_0 \subseteq W_1, ..., W_t \subseteq V$ in the following manner: $W_0$ contains the tree median of $T$, and then to create $W_i$ from $W_{i-1}$ we simply add to $W_i$ the tree medians of every compartment in $\mathcal{C}_T(W_i)$.

For each $i$, we only make oracle calls that take the form $f(V_{-v}(u), s)$ or $f(V_{-v}(u) \cup \{v\}, s)$. We avoid choices of $(u, v)$ that will cause evaluation on overlapping sets, based on information gained on processing $W_{i-1}$ in the same way. In this way we only make essentially $O(1)$ 'amortized' calls to the oracle for each $i$. For details see the full paper.

After removing nodes via the reaching criterion, we only need to do this on a subtree of $T$, which we can assume takes the same time as on the full tree. One can see that $t = O(\log n)$ thus the peaking criterion takes at most $O(k \log n)$ amortized oracle calls.

### 3.4.3 Reaching criterion by Binary Search

Intuitively, with the reaching criterion we look for an edge $(u, v)$ in $T_{H(S)}$ so that $T_{-v}(u)$, which contains at least one sink, can be removed. Now given *adjacent* hubs $h_1$ and $h_2$, i.e. hubs where there are no other hubs along the path from $h_1$ to $h_2$, consider any subtree of $T$ rooted at $h_1$, in which $h_2$ is a descendent of $h_1$. Then exactly one of the following is true:

**(P1)** There is an edge $(u, v)$ in the path $\Pi(h_1, h_2)$ between $h_1$ and $h_2$, where $u$ is a child of $v \neq h_1$, such that $T_{-v}(u)$ is recursively self-sufficient, but the subtree rooted at $v$ is not.

**(P2)** Let $u$ be the child of $h_1$ that is on the path between $h_1$ and $h_2$. Then the subtree rooted at $u$, i.e. $T_{-h_1}(u)$, is recursively self-sufficient.

As $h_1$ and $h_2$ are adjacent hubs, for any edge $(u, v)$ along the path, where $v \neq h_1$ is the parent of $u$, the subtree rooted at $v$ is recursively self sufficient only if the subtree rooted at $u$ is. Suppose we know that the subtree rooted at $h_2$ is recursively self-sufficient.

In the iterative algorithm we move upwards from $h_2$ to $h_1$ gradually until we find such an edge, or upon reaching $h_1$; this can be replaced by a binary search. This idea will let us only use $O(k^2 \log n)$ calls; proper amortization with pruning can reduce this to $O(k \log n)$ oracle calls. See full paper. Theorem 2 follows from the above faster algorithm.

## 4 Full problem: cost minimization

Given an algorithm for the bounded cost problem, we may perform a binary search over possible values of $\mathcal{T}$ for the minimal $\mathcal{T}^*$ allowing evacuation with $k$ sinks. To produce a *strongly* polynomial time algorithm, at a higher level, we wish to search among a finite, discrete set of possible values for $\mathcal{T}^*$. This can be done by a *parametric searching* technique.

## 4.1 Iterative approach

We start by modifying the iterative algorithm for bounded cost. In that algorithm, the specific value of $\mathcal{T}$ dictates the contents of $T$, $S$, $S_{\text{out}}$ etc., as well as which node pairs satisfy either of PC and RC, at each step of that depends upon the outcome of a comparison of the form $f(\cdot,\cdot) \le \mathcal{T}$, obtained from calling the oracle.

The idea is to run a parametric search version of the bounded cost algorithm. $\mathcal{T}$ will no longer be a constant; we *interfere* the normal course of the algorithm by changing $\mathcal{T}$ during runtime. The decision to interfere is based on a *threshold margin* $(\mathcal{T}^L, \mathcal{T}^H]$ that we maintain, to keep track of candidate values of $\mathcal{T}^*$. Initially, $(\mathcal{T}^L, \mathcal{T}^H] = (-\infty. + \infty]$, and $\mathcal{T} = 0$.

The following process terminates with some $\mathcal{T} \in (\mathcal{T}^L, \mathcal{T}^H]$; call this the 'bounded cost algorithm with interference'. Every time we evaluate $a = f(\cdot,\cdot)$, we set $\mathcal{T}$ based on the following, before making the comparison $a \le \mathcal{T}$ and proceeding with the relevant if-clause.

1. If $a \le \mathcal{T}^L$, set $\mathcal{T} = \mathcal{T}^L$, so the if-clause always resolves as $f(\cdot,\cdot) \le \mathcal{T}$.
2. If $a > \mathcal{T}^H$, set $\mathcal{T} = \mathcal{T}^H$, so the if-clause always resolves as $f(\cdot,\cdot) > \mathcal{T}$.
3. If $a \in (\mathcal{T}^L, \mathcal{T}^H]$, run a separate clean, non-interfered instance of the bounded cost algorithm with threshold value $\mathcal{T} := a$, and observe the output.
   - Output is 'No': set $\mathcal{T}^L := a$, and $\mathcal{T} := a$, resolving the if-clause as $a = f(\cdot,\cdot) \le \mathcal{T} = a$.
   - Otherwise, set $\mathcal{T}^H := a$, and $\mathcal{T} := \mathcal{T}^L$.

▶ **Lemma 22.** *Let $(\mathcal{T}_<, \mathcal{T}_>]$ be the threshold margin at the end of the bounded cost algorithm with interference. Then $\mathcal{T}_> = \mathcal{T}^*$. In particular, we can then run the bounded cost algorithm (non-interfered) on $\mathcal{T} := \mathcal{T}_>$ to retrieve the optimal feasible configuration.*

▶ **Theorem 23.** *Minmax tree facility location can be solved in $O(n^2)$ calls to $\mathcal{A}$.*

**Proof.** We always allow the interfered algorithm to make progress, albeit with changing values of $\mathcal{T}$, so Lemma 20 still applies; $f(\cdot,\cdot)$ is evaluated at most $O(n)$ times in the interfered algorithm, thus we also launch a separate instance of the feasibility test $O(n)$ times. ◀

## 4.2 Using divide-and-conquer and binary search

The above idea still works for applying RC, that we can use the same ideas of calling the feasibility test and interfering as we evaluate $f(\cdot,\cdot)$. Thus we only interfere $O(k \log n)$ times, making $O(k^2 \log^2 n)$ total calls to the oracle.

But it does not work well with the peaking criterion; that the divide-and-conquer algorithm for the peaking criterion relies very strongly on amortization, and a naive application of interference will perform $O(n)$ feasibility tests, while we aim for $O(k \log n)$.

The basic idea is to filter through values of $f(\cdot,\cdot)$ where we decide to interfere. Intuitively, the divide and conquer algorithm can be organized in $t$ layers in reference to $W_1, ..., W_t$ where $t = O(\log n)$, for each we evaluate $f(\cdot,\cdot)$ on certain pairs of nodes and sets. Each evaluation of $f(\cdot,\cdot)$ can be identified with an edge of $T$, thus in each layer we have at most $O(n)$ evaluations, producing a list of $O(n)$ values.

Thus, at each layer we evaluate $f(\cdot,\cdot)$, and binary search for a pair of values $a_<, a_>$ such that $a_< \le \mathcal{T}^* < a_>$, making $O(\log n)$ calls to the bounded-cost algorithm, and then set $\mathcal{T} = a_<$ when proceeding to mark nodes and place sinks, before moving to the next layer.

This gives $O(\log^2 n)$ calls for a single application of the peaking criterion. As we only need to apply the peaking criterion $O(k)$ times, the resulting number of calls to the feasibility test is $O(k \log^2 n)$. As each call takes $O(nk \log^3 n)$ time, Theorem 1 then follows.

## 5 Conclusion

Given a Dynamic flow network on a tree $T_{\text{in}} = (V, E)$ we derive an algorithm for finding the locations of $k$ sinks that minimize the maximum time needed to evacuate the entire graph. Evacuation is modelled using dynamic confluent flows. Only an $O(n \log^2 n)$ time algorithm for solving the one-sink ($k = 1$) case was previously known.

This paper gives the first polynomial time algorithm for solving the arbitrary $k$-sink problem, developed in two parts. Section 3 gives an $O(nk \log^3 n)$ algorithm to test the feasibility of completing evacuation in time $\mathcal{T}$ with $k$ sinks. Section 4 showed how to modify this to an $O(nk^2 \log^5 n)$ algorithm for finding the minimum such $\mathcal{T}$ that permits evacuation.

## References

1 J. E. Aronson. A survey of dynamic network flows. *Annals of Operations Research*, 20(1):1–66, 1989. doi:10.1007/BF02216922.

2 G. P. Arumugam, J. Augustine, M. J. Golin, and P. Srikanthan. Optimal evacuation on dynamic paths with general capacities of edges. *Unpublished Manuscript*, 2015.

3 J. Chen, R. Rajaraman, and R. Sundaram. Meet and merge: Approximation algorithms for confluent flows. *Journal of Computer and System Sciences*, 72(3):468–489, 2006.

4 Jiangzhuo Chen, Robert D berg, László Lovász, Rajmohan Rajaraman, Ravi Sundaram, and Adrian Vetta. (Almost) Tight bounds and existence theorems for single-commodity confluent flows. *Journal of the ACM*, 54(4), jul 2007.

5 Daniel Dressler and Martin Strehler. Capacitated Confluent Flows: Complexity and Algorithms. In *7th International Conference on Algorithms and Complexity (CIAC'10)*, pages 347–358, 2010.

6 Lisa Fleischer and Martin Skutella. Quickest Flows Over Time. *SIAM Journal on Computing*, 36(6):1600–1630, January 2007.

7 L. R. Ford and D. R. Fulkerson. Constructing Maximal Dynamic Flows from Static Flows. *Operations Research*, 6(3):419–433, June 1958.

8 Greg N Frederickson. Parametric search and locating supply centers in trees. In *Proceedings of the Second Workshop on Algorithms and Data Structures (WADS'91)*, pages 299–319. Springer, 1991.

9 Michael R. Garey and David S. Johnson. *Computers and intractability: A Guide to the Theory of NP-Completeness*. W.H. Freeman and Company, 1979.

10 Yuya Higashikawa, M. J. Golin, and Naoki Katoh. Minimax Regret Sink Location Problem in Dynamic Tree Networks with Uniform Capacity. In *The 8th International Workshop on Algorithms and Computation (WALCOM'2014)*, pages 125–137, 2014. doi:10.1007/978-3-319-04657-0_14.

11 Yuya Higashikawa, Mordecai J Golin, and Naoki Katoh. Multiple sink location problems in dynamic path networks. *Theoretical Computer Science*, 607:2–15, 2015.

12 B Hoppe and É Tardos. The quickest transshipment problem. *Mathematics of Operations Research*, 25(1):36–62, 2000.

13 N. Kamiyama, N. Katoh, and A. Takizawa. Theoretical and Practical Issues of Evacuation Planning in Urban Areas. In *The Eighth Hellenic European Research on Computer Mathematics and its Applications Conference (HERCMA2007)*, pages 49–50, 2007.

14 Satoko Mamada and Kazuhisa Makino. An Evacuation Problem in Tree Dynamic Networks with Multiple Exits. In Tatsuo Arai, Shigeru Yamamoto, and Kazuhi Makino, editors, *Systems & Human Science-For Safety, Security, and Dependability; Selected Papers of the 1st International Symposium SSR2003*, pages 517–526. Elsevier B.V, 2005.

15 Satoko Mamada, Takeaki Uno, Kazuhisa Makino, and Satoru Fujishige. A tree partitioning problem arising from an evacuation problem in tree dynamic networks. *Journal of the Operations Research Society of Japan*, 48(3):196–206, 2005.

16 Satoko Mamada, Takeaki Uno, Kazuhisa Makino, and Satoru Fujishige. An $O(n \log^2 n)$algorithm for the optimal sink location problem in dynamic tree networks. *Discrete Applied Mathematics*, 154(2387-2401):251–264, 2006.

17 M. M. B. Pascoal, M. Eugénia V. Captivo, and J. C. N. Clímaco. A comprehensive survey on the quickest path problem. *Annals of Operations Research*, 147(1):5–21, August 2006.

18 F. Bruce Shepherd and Adrian Vetta. The Inapproximability of Maximum Single-Sink Unsplittable, Priority and Confluent Flow Problems, 2015. `arXiv:1504.0627`.

19 Martin Skutella. An introduction to network flows over time. In William Cook, László Lovász, and Jens Vygen, editors, *Research Trends in Combinatorial Optimization*, pages 451–482. Springer, 2009.