Minmax Centered $k$-Partitioning of Trees and Applications to Sink Evacuation with Dynamic Confluent Flows

Di Chen · Mordecai Golin

Received: 29 March 2018 / Accepted: 6 December 2022 / Published online: 29 December 2022
© The Author(s), under exclusive licence to Springer Science+Business Media, LLC, part of Springer Nature 2022

Abstract
Let $T = (V, E)$ be a tree with associated costs on its subtrees. A minmax $k$-partition of $T$ is a partition into $k$ subtrees, minimizing the maximum cost of a subtree over all possible partitions. In the centered version of the problem, the cost of a subtree is defined as the minimum cost of “servicing” that subtree using a center located within it. The problem motivating this work was the sink-evacuation problem on trees, i.e., finding a collection of $k$-sinks that minimize the time required by a confluent dynamic network flow to evacuate all supplies to sinks. This paper provides the first polynomial-time algorithm for solving this problem, running in $O\left(\max(k \log k, \log n)k^2 n \log^4 n\right)$ time. The technique developed can be used to solve any Minmax Centered $k$-Partitioning problem on trees in which the servicing costs satisfy some very general conditions. Solutions can be found for both the discrete case, in which centers must be on vertices, and the continuous case, in which centers may also be placed on edges. The technique developed also improves previous results for solving the sink evacuation problem on a tree, given the location of the sinks in advance.

Keywords Sink evacuation · Dynamic flows · Confluent flows · Facility location · Parametric search · Tree partitioning · Tree centroid

1 Introduction

The main result of this paper is the derivation of a new method for solving the general minmax centered $k$-partitioning problem on trees. The initial motivation was the con-
A 6-partition $P = \{P_1, \ldots, P_6\}$ of tree $T$. Component $P_i$ has associated sink or center $s_i$. $f(P_i, s_i)$ is the cost of servicing $P_i$ using $s_i$. $f_S(P, S) = \max_{1 \leq i \leq 6} f(P_i, s_i)$ is the full cost of servicing $T$ with partition $P$ and sink set $S = \{s_1, \ldots, s_6\}$.

Fig. 1 A 6-partition $\mathcal{P} = \{P_1, \ldots, P_6\}$, of tree $T$. Component $P_i$ has associated sink or center $s_i$. $f(P_i, s_i)$ is the cost of servicing $P_i$ using $s_i$. $f_S(P, S) = \max_{1 \leq i \leq 6} f(P_i, s_i)$ is the full cost of servicing $T$ with partition $P$ and sink set $S = \{s_1, \ldots, s_6\}$.

A $k$-partition of a tree $T = (V, E)$ is the removal of $k - 1$ edges to create $k$ subtrees. Let $f(P)$ denote the cost of subtree $P \subseteq V$ (subtrees will be denoted by their nodes). The cost of partition $\mathcal{P} = \{P_1, \ldots, P_k\}$ is $F(\mathcal{P}) = \max_i f(P_i)$. The minmax $k$-partition problem is to find a $k$-partition $\mathcal{P}$ of $T$ that minimizes $F(\mathcal{P})$.

$f(P)$ may sometimes be further defined as $f(P, s)$, the cost of servicing the subtree from some sink or center $s \in P$. The cost of the partition will then be $F(\mathcal{P}, S) = \max_i f(P_i, s_i)$ where $S = \{s_1, \ldots, s_k\}$ and $s_i \in P_i$. See Fig. 1. The minmax centered $k$-partition problem is to find $\mathcal{P}, S$ that minimizes $F(\mathcal{P}, S)$.

Becker et al. [6] introduced a shifting algorithm for constructing minmax partitions of trees when $f(P)$ is the sum of the weights of the nodes in $P$. This technique was then improved and generalized to other functions by them and other authors [2, 4, 5, 35]. [26, 27] discuss extensions to centered partitions. These results only hold for the very restrictive class of Invariant functions $f(P)$ (see [5] for a definition). In particular, the QFP cost that will interest us and be defined below will not be an invariant function.

If all nodes $v \in V$ have given weights $w_v$ and $d(v, s)$ is the path-length distance from $v$ to $s$, then $f(P, s) = \max_{v \in P} w_v d(v, s)$ defines the $k$-center problem, which has its own separate literature. Frederickson [18] gives an $O(n)$ algorithm for $k$-center in an unweighted tree, i.e., $w_v \equiv 1$, while the weighted case can be solved in $O(n \log n)$ time [38].

The problem motivating this paper arises from evacuation using Dynamic Confluent Flows. 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 $\tau_e$ – the time required to traverse it – and a capacity $c_e$, limiting how much flow can enter the edge in one time unit. If all edges have the
same capacity \( c_e = c \) the network has uniform capacity. As supplies move around the graph, congestion can occur as supplies back up waiting to enter a vertex, increasing the time needed to send a flow.

Dynamic flow networks were introduced by Ford and Fulkerson [17] and have since been extensively used and analyzed. The Quickest Flow Problem (QFP) starts with \( w_v \) units of flow on (source) node \( v \) and asks how quickly all of this flow can be moved to designated sinks. Good surveys of the problem and applications can be found in [3, 15, 34, 37].

One variant of the QFP is the transportation problem in which each sink has a specified demand with total source availability equal to total demand requirement. The problem is to find the minimum time required to satisfy all of the demands. The first polynomial time algorithm for that problem was given by [23] with later improvements by [16].

A variant of the QFP can also model evacuation problems, see e.g., [21] for a history. In this, vertex supplies can be visualized as people in one or multiple buildings and the problem is to find a routing strategy (evacuation plan) that evacuates all of them to specified sinks (exits) in minimum time. This differs from the transportation problem in that the problem is to fully evacuate the sources, not to satisfy the sinks; sinks do not have predefined demands and may absorb arbitrarily large units of supply.

An optimal solution to this problem could assign different paths to different units of supply starting from the same vertex. Physically, this could correspond to two people starting from the same location travelling radically different evacuation paths, possibly even to different exits.

A constrained version of the problem, the one addressed here, is for the plan to assign to each vertex \( v \) exactly one evacuation edge, \( e_v = (v, u_v) \), i.e., a sign stating “this way out”. All people starting at or passing through \( v \) must evacuate through \( e_v \). After arriving at \( u_v \) they continue onto \( u_v \)'s unique evacuation edge \( e_{uv} \). They continue following these unique evacuation edges until reaching a sink, where they exit. The initial problem is, given the sinks, to determine a plan minimizing the maximum time needed to evacuate everyone. Note that if each \( v \) has a unique evacuation edge \( e_v \), then the \( e_v \) must form a directed forest with the sinks being the roots of the trees. Thus, an evacuation plan of tree \( T \) using \( k \) sinks is a centered \( k \)-partition of \( T \). See Fig. 2. A different version of the problem 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.

Flows with the property that all flows entering a vertex leave along the same edge are known as confluent; even in the static case constructing an optimal confluent flow in a general graph \( G \) is known to be very difficult. If \( P \neq \text{NP} \), then it is impossible to construct a constant-factor approximate optimal confluent flow in polynomial time on a general graph [11, 12, 14, 36] even with only one sink.

If edge capacities are “large enough” then no congestion occurs and every person starting at node \( v \) should follow the same shortest path it can to an exit. The cost of the plan will be the length of the maximum shortest path. Minimizing this is is

---

1 Confluent flows occur naturally in problems other than evacuations, e.g., packet forwarding and railway scheduling [14].
Fig. 2 Each vertex except for the $s_j$ has a unique associated evacuation edge (the $s_j$ evacuate to themselves). These edges form a forest of directed in-trees with roots $s_j$. This forest defines a centered 5-partition of $T$ with centers $s_j$. The cost of the partition will be the maximum time required for a node to evacuate to its assigned exit $s_j$. Flows can merge and cause congestion so this evacuation time is a function of entire subtrees and not just of individual node-sink pairs exactly the $k$-center problem on graphs which is already known to be NP-Hard [19, ND50]. Unlike $k$-center, which is polynomial-time solvable for fixed $k$, Kamiyama et al. [24] proves by reduction to Partition that finding the min-time evacuation protocol is NP-Hard for general graphs even when restricting $k = 1$. This was later extended [20] to show that for $k = 1$ and the sink location fixed in advance, it is still impossible to approximate the QFP time to within a factor of $o(\log n)$ if $P \neq NP$.

The only solvable known case for the sink location problem for general $k$ is for $G$ a path [7]. For paths with uniform capacities this runs in $\min(O(n + k^2 \log^2 n), O(n \log n))$ time; for paths with general capacities in $\min(O(n \log n + k^2 \log^4 n), O(n \log^3 n))$ time.

When $G$ is a tree, the 1-sink location problem can be solved [30] in $O(n \log^2 n)$ time. This can be reduced [8, 21] down to $O(n \log n)$ for the uniform capacity version, i.e., all the $c_e$ are identical. If the locations of the $k$ sinks are given as input, [28] gives a $O(n(c \log n)^{k+1})$ time algorithm evacuation protocol, where $c$ is some constant. This is the problem of partitioning the tree optimally, given that the centers are already known. For “large” $k$, [29] reduced the time down to $O(n^2 k \log^2 n)$. The literature does not contain any algorithm for solving the sink-location problem on trees. The best solution using current known results would be to try all possible $\Theta(n^{k-1})$ decompositions of the tree into $k$ subtrees and apply the algorithm of [30], yielding $O(n^k \log^2 n)$ time.

When $k = 1$, [28] also provides an $O(n \log^2 n)$ algorithm for calculating the evacuation cost to a single known sink. For the uniform capacity case, [22, p. 34] gives a formula that reduces the calculation time down to $O(n \log n)$. These two calculation algorithms will be used as oracles in the sequel.

The discussion above implicitly assumed that the sinks must be vertices of the original graph. This is known as the discrete case. Another possibility would be to
permit sinks to be located anywhere, on edges as well as vertices. This variation is known as the continuous case.

Historically, this distinction is also explicit in the $k$-center in a tree literature. More specifically, Frederickson’s [18] $O(n)$ algorithm for $k$-center in an unweighted tree worked in both the continuous and discrete cases. For weighted $k$-center, though, the two cases needed two different sets of techniques. [33] gave an $O(n \log^2 n)$ algorithm for the discrete case while the continuous case required $O(n \log^2 n \log \log n)$ time [32]. It was only later realized that a parametric searching technique [13] could reduce the continuous case down to $O(n \log^2 n)$ as well. A very recent result [38] has reduced the time for both down to $O(n \log n)$. Much earlier work noted that weighted $k$-center restricted to the line can be solved in $O(n \log n)$ in both the discrete and continuous cases; these were also originally solved separately; [10] provides a good discussion of the history of that problem.

1.1 Our Contributions

This paper gives the first polynomial time algorithm for solving the $k$-sink location problem on trees. It uses as an oracle a known algorithm for calculating the cost of the problem when $k = 1$ and the sink is known in advance. Our results will be applicable to both the discrete and continuous versions of the problem.

**Theorem 1** *The $k$-sink location problem for sink-evacuation can be solved in
- $O(\max(k \log k, \log n) k^2 n \log^4 n)$ time for general-capacity edges and
- $O(\max(k \log k, \log n) k^2 n \log^3 n)$ time for uniform-capacity edges.*

This result will be a special case of a general technique that works for a large variety of minmax cost functions on trees. Section 2 formally defines the Sink-Evacuation problem on trees, the more general class of functions for which our technique works, and then states our results.

It is instructive to compare our approach to Frederickson’s [18] $O(n)$ algorithm for solving the unweighted $k$-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.

The main difficulty in solving the sink-evacuation problem is that no polynomial time feasibility test for $k$-sink evacuation on trees was previously known. The majority of this paper is devoted to constructing such a test. Section 3 derives useful properties of the feasibility problem and Sect. 4 utilizes these properties to construct a testing algorithm. This algorithm works by making $O(k^2 \log n)$ (amortized) calls to the 1 fixed-sink algorithm oracle.

There is also no small set of easily defined cost values known to contain the optimal solution. We sidestep this issue in Sect. 5 by doing parametric searching within our feasibility testing algorithm, leading to Theorem 1.
Fig. 3 An illustration of evacuation of a tree to sink, s. Initial values $w_v$ are above $v$. Each edge $e$ is labelled with a (capacity, length) pair $(c_e, \tau_e)$. The goal is to evacuate all supplies to $s$. Note that the tree contains 3 different branches containing, respectively, $\{a\}$, $\{b, c\}$ and $\{d, e, f, g, h, i\}$ whose evacuation times can be calculated separately. The time required to evacuate all supplies to $s$ is 22, which is when the last supply from $i$ arrives at $s$.

Sections 3, 4 and 5 assume the discrete version of the problem. Section 6 describes the modifications necessary to extend the algorithm to work in the continuous case.

In Sect. 7 we conclude by noting that a slight modification to the algorithm also allows solving the $k$-sink evacuation problem on trees, i.e., the case when the sink locations are known in advance, in $O(\max(k \log k, \log n) k^2 n \log^4 n)$ time. This improves, for $k \ll \sqrt{n}$, the best previously known $O(n^2 k \log^2 n)$ [29] for solving the problem.

2 Definitions and Results

2.1 Sink Evacuation

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 flow to enter edge $e$ per every unit of time.

Consider $w_u$ units of (supply) flow waiting at vertex $u$ at time $t = 0$ to traverse edge $e = (u, v)$. They enter $e$ at a rate of $c_e$ units of flow per unit time so the last flow enters $e$ at time $w_u/c_e$. This flow then travels another $\tau_e$ time to reach $v$. The total time required to move all flow from $u$ to $v$ is then $w_u/c_e + \tau_e$.

If two edges were combined in a path from $u \rightarrow v \rightarrow s$ then flow from $u$ travelling to $s$ might have to wait at $v$ for all the $w_v$ flow to first enter $(v, s)$. When multiple paths meet, this results in congestion that can delay evacuation time in strange ways.

Figure 3 illustrates different types of congestion and gives an example of calculating the evacuation time of a tree to a given sink.

Given a graph $G$, distinguish a subset $S \subseteq V$ with $|S| = k$ as sinks (exits). An evacuation plan specifies, for each vertex $v \notin S$, the unique edge along which all flow
starting at or passing through \( v \) evacuates. Furthermore, starting at any \( v \) and following the edges will lead from \( v \) to one of the \( S \) (if \( v \in S \), flow at \( v \) evacuates immediately through the exit at \( v \)). As noted earlier (Fig. 2) the evacuation plan defines a confluent flow. The evacuation edges form a directed forest; the root of each tree is one of the designated sinks in \( S \).

Given evacuation plan \( P \) and the \( w_v \) specifying the initial flow supply starting at each node, one can calculate, for each vertex, the time (with congestion) required for all of its flow supply to evacuate. The maximum of this over all \( v \) is the minimum time required to required to evacuate all items to some exit using the rules above. Call this the cost for \( S \) associated with the evacuation plan and denote it by \( f(P, S) \). Fig. 3 provides an illustration.

The \( k \)-sink location problem is to find a subset \( S \) of size \( k \) and associated \( P \) that minimizes \( f(P, S) \).

### 2.2 General Problem Formulation

The input will be a tree \( T_{in} = (V_{in}, E_{in}) \), and a positive integer \( k \). Set \( n = |V_{in}| = |E_{in}| + 1 \). The output will be \( S \subseteq V_{in}, |S| \leq k \), and an associated partition \( P \) of \( T_{in} \) into \( |S| \) subtrees, each containing one vertex in \( S \), that minimizes \( f(P, S) \) over all possible such pairs.

The algorithms will not explicitly deal with the complicated mechanics of evacuation calculations. Instead they will solve the location problem for any minmax monotone cost \( f(\cdot, \cdot) \), given an oracle for solving a one-sink problem in which the location of the sink is pre-specified.

This level of abstraction simplifies the formulation and understanding of the algorithms. It can also be useful for solving other similar problems.

#### 2.2.1 Minmax Monotone Cost Functions

Minmax monotone cost functions are defined below. Note that this definition is consistent with the specific properties of the evacuation problem.

**Definition 1** Let \( F = (V, E) \) be a forest.

Let \( U \subseteq V \). “\( U \) is a subtree of \( T \)” means that \( U \) induces a subtree of \( F \).

For any \( u \in V \), \( \Gamma(u) = \{ v \in V : (u, v) \in E \} \) are the neighbors of \( u \) in \( V \).

For \( U \subseteq V \), \( \Gamma(U) = \bigcup_{u \in U} \Gamma(u) \) are the neighbors of \( U \) in \( V \).

A Partition of \( V \) is \( P = \{ P_1, P_2, \ldots, P_t \} \) such that each \( P_i \) is a subtree, \( \cup_i P_i = V \), and \( \forall i \neq j, P_i \cap P_j = \emptyset \). \( P_i \) are the blocks of \( P \).

Let \( S = \{ s_1, s_2, \ldots, s_t \} \subseteq V \). \( \Lambda[S, V] \) denotes the set of all partitions \( P = \{ P_{s_1}, P_{s_2}, \ldots, P_{s_t} \} \) of \( V \) such that \( \forall i, S \cap P_{s_i} = \{ s_i \} \).

A sink configuration \((S, P)\) is a pair \( S \subseteq V \) and associated \( P \in \Lambda[S, V] \).

Let \( f : 2^V \times V \rightarrow [0, +\infty] \) be an atomic cost function. \( f(P, s) \) can be interpreted as the cost for sink \( s \) to serve the tree induced by \( P \). This interpretation imposes the following natural constraints:

1. For \( U \subseteq V, s \in V \), ...
Fig. 4 Example of max-composition. Let $U$ denote the complete tree. Removing $s$ creates a forest with three trees, $U_1, U_2, U_3$. By definition,

$$f(U, s) = \max\{f(U_1 \cup \{s\}, s), f(U_2 \cup \{s\}, s), f(U_3 \cup \{s\}, s)\}$$

- if $U = \{s\}$, then $f(U, s) = 0$.
- if $U$ is not a subtree or $s \notin U$ then $f(U, s) = +\infty$.

2. Set monotonicity
If $s \in U_1 \subseteq U_2 \subseteq V$, then $f(U_1, s) \leq f(U_2, s)$.

3. Path monotonicity
Let $U \subset V, s \notin U$ but $s \in \Gamma(u)$ for some $u \in U$. Then $f(U \cup \{s\}, s) \geq f(U, u)$. Intuitively, this means that as a sink serving $U$ moves away from $U$, the cost of servicing $U$ can not decrease.

4. Max tree composition (Fig. 4)
Let $T = (U, E')$ be a subtree of $F$ and $s \in U$ a node with $t$ neighbors. Set $\mathcal{F} = \{T_1, \ldots, T_t\}$ to be the forest created by removing $s$ from $T$, and $U_1, \ldots, U_t$ the respective vertices of each tree in $\mathcal{F}$. Then

$$f(U, s) = \max_{1 \leq i \leq t} f(U_i \cup \{s\}, s).$$

The subtrees $U_i$ will be called slices of $U$ defined by $s$.

Note that 1-4 define a cost function over one subtree and one sink. Function $f(\cdot, \cdot)$ is now naturally extended to work on on partitions and sets (Fig. 1).

5. Max partition composition

$$\forall \mathcal{P} \in \Lambda[S, V], \quad f(\mathcal{P}, S) = \max_{s_i \in S} f(\mathcal{P}_{s_i}, s_i). \quad (1)$$

**Definition 2** A cost function $f(\mathcal{P}, S)$ that satisfies properties 1-5 is called minmax monotone.

Given $k > 0$, the main problem will be to find sink configuration $(S^*, \mathcal{P}^*)$ such that $\mathcal{P}^* \in \Lambda[S^*, V_{\text{in}}]$ and

$$f(\mathcal{P}^*, S^*) = \min_{S \subseteq V, |S| \leq k, \mathcal{P} \in \Lambda[S, V_{\text{in}}]} f(\mathcal{P}, S). \quad (2)$$

Our algorithms make calls directly to an oracle $\mathcal{A}$ that, given subtree $U$ of $T_{\text{in}}$ and $v \in U$, computes $f(U, v)$. As mentioned, in our case of interest, [28] provides an $O(n \log^2 n)$ oracle for general-capacity sink evacuation and [22, p. 34] provides $O(n \log n)$ oracle for uniform-capacity sink evacuation.

Finally, later amortization arguments will require the following definition:
**Definition 3** If $A$ runs in time $t_A(n)$, then $A$ is asymptotically subadditive if

- $t_A(n) = \Omega(n)$ and is non-decreasing.
- For all nonnegative $n_i$, $\sum_i t_A(n_i) = O\left(t_A\left(\sum_i n_i\right)\right)$.
- $t_A(n + 1) = O\left(t_A(n)\right)$

For any constants $x \geq 1$ and $y \geq 0$, any function of the form $n^x \log^y n$ is asymptotically subadditive. In particular, the oracles mentioned above are asymptotically subadditive.

### 2.3 Results

The remainder of the paper is devoted to deriving three algorithms.

The first algorithm and the majority of the paper, provides a feasibility test, which solves a simplified, bounded-cost version of the problem. Given $k$ and $T$, determine whether there exists a $k$-partition with cost at most $T$.

**Problem Bounded cost minmax $k$-sink**

- **Input**
  - $T_{in} = (V_{in}, E_{in})$, $k \geq 1$, $T \geq 0$
  - $S_{out} \subseteq V_{in}$ and $P_{out} \in A[S_{out}, V_{in}]$ s.t. $|S_{out}| \leq k$ and $f(P_{out}, S_{out}) \leq T$.
- **Output**
  - If such a $(S_{out}, P_{out})$ pair does not exist, output 'No'.

The second algorithm is for the original general problem. To find the location of $k$ sinks that minimize the cost of a $k$-partition.

**Problem Minmax $k$-sink location**

- **Input**
  - $T_{in} = (V_{in}, E_{in})$, $k \geq 1$
- **Output**
  - $S_{out} \subseteq V_{in}$ satisfying $|S| \leq k$ and $P_{out} \in A[S_{out}, V_{in}]$ satisfying Eq. (2)

Our first result is

**Theorem 2** If $A$ is an asymptotically subadditive algorithm for solving the fixed 1-sink problem that runs in $t_A(n)$ time, then the bounded cost minmax $k$-sink problem can be solved in time $O(k^2 t_A(n) \log n)$.

Combining this algorithm with a careful application of parametric searching will yield a solution to the general problem:

**Theorem 3** If $A$ is an asymptotically subadditive algorithm for solving the fixed 1-sink problem that runs in $t_A(n)$ time then the minmax $k$-sink problem can be solved in time $O\left(\max(k \log k, \log n) k^2 t_A(n) \log^2 n\right)$.

Theorem 1 follows directly from this and the 1-sink oracles given by [28] and [22, p. 34].
A simplification of the 2nd algorithm will also, in the same time, solve the specialized partitioning version in which the sinks are fixed in advance. We will call a minmax monotone function \textit{relaxed} if the defining \( f(\cdot, \cdot) \) satisfies properties 1, 2, and 4 from Sect. 2.2.1 but does not necessarily satisfy property 3 (path monotonicity).\(^2\)

| Problem | Relaxed Minmax \( k \) fixed-sink |
|---------|----------------------------------|
| Input   | Tree \( T_{in} = (V_{in}, E_{in}) \), \( S \subseteq V_{in} \), \( |S| = k \) |
| Output  | \( \mathcal{P}_{out} \in \Lambda[S, V_{in}] \) s.t. \( f(\mathcal{P}_{out}, S) = \min_{\mathcal{P} \in \Lambda[S, V_{in}]} f(\mathcal{P}, S) \) |

\textbf{Theorem 4} If \( A \) is an asymptotically subadditive algorithm for solving the fixed relaxed 1-sink problem that runs in \( t_A(n) \) time and that further satisfies \( t_A(2n) = O(t_A(n)) \), then the minmax \( k \) fixed-sink problem can be solved in time \( O\left( \max(k \log k, \log n) k^2 t_A(n) \log^2 n \right) \).

For the sink evacuation problem, plugging the \( O(n \log^2 n) \) oracle into Theorem 4 leads to a \( O\left( k^2 n \log^4 n \max(k \log k, \log n) \right) \) time algorithm, improving upon the previously known \( O(n(c \log n)^k + 1) \) [28] and \( O(n^2 k \log^2 n) \) [29] algorithms when \( 5 < k \ll \sqrt{n} \).

Theorems 2 and 3 are technically only stated for the discrete version, in which \( S_{out} \subseteq V_{in} \). They also hold in the continuous version. This is proven in Sect. 6, which also introduces the extra notation needed for this case.

\section*{2.4 More Applications}

Although our algorithm was motivated by confluent dynamic flows it is surprisingly easy to apply to unrelated problems. We provide three examples below. The input is always a tree \( T_{in} = (V_{in}, E_{in}) \).

Given a tree \( U \) and \( s \in U \), recall the definition of \textit{slices} \( U_i \) in the max-composition rule. The \( U_i \) were the subtrees that resulted by removing \( s \).

\textbf{Example 1: Weighted} \( k \)-center

Each vertex has weight \( w_v \) and each edge \((u, v)\) has length \( d(u, v) \). For any pair \((u', v') \notin E \), \( d(u', v') \) is the sum of the lengths of the edges on the unique path connecting \((u', v')\).

As a warm-up application we note that our algorithm immediately yields a (non-optimal) algorithm for weighted \( k \) center by setting

\[ f(U \cup \{s\}, s) = \max_{u \in U} w_u d(s, u) \]

where \( U \) is a subtree, \( s \notin U \) but \( s \in \Gamma(U) \). This \( f(\cdot, \cdot) \) satisfies the minmax monotone cost function properties laid out in Sect. 2.2.1 and can be evaluated in

\(^2\) Because the sinks are predefined, they never move and path monotonicity is superfluous.
$O(|U|)$ time using a breadth-first search scan of the tree. Thus Theorem 3 yields an $O(\max(k \log k, \log n)k^2n \log^2 n)$ time algorithm for solving the weighted $k$-center problem.

The algorithm above is slower than the $O(n \log n)$ algorithm of [38]. But, that algorithm strongly use parametric searching in a polynomially bounded space (costs are defined by pairs of vertices, so there are only $O(n^2)$ possible solutions in the discrete case). It would be difficult to modify them to include general constraints. As illustrated below, Theorem 3 permits adding many types of constraints without any increase in running time.

Example 2: Weight constrained weighted $k$-center
Now denote the weight of a subtree $U \subseteq V_{in}$ by $W(U) = \sum_{u \in U} w_u$.

Consider the following combination of the weighted $k$-center problem and minmax weight-partitioning problem [6] that adds the constraint that the weight of all slices is at most some fixed threshold $W > 0$. This $W$ can be viewed as a natural limit on the capacity of the service center $s$.

For $U$ is a subtree, $s \notin U$ but $s \in \Gamma(U)$ set

$$f(U \cup \{s\}, s) = \begin{cases} \max_{u \in U} w_ud(s, u) & \text{if } W(U) \leq W, \\ \infty & \text{Otherwise} \end{cases}$$

This function also satisfies the minmax monotone function properties and can still easily be evaluated by a breadth-first search scan of the tree in $O(|U|)$ time. Solving the minmax $k$-sink problem for this $f(\cdot, \cdot)$ function using Theorem 3 exactly solves the weighted $k$-center problem in which each slice is constrained to have weight at most $W$ in $O(\max(k \log k, \log n)k^2n \log^2 n)$ time.

Adding additional constraints is not difficult. If $d_H(u, v)$ is defined to be the number of edges (hop distance) on the path connecting $u$ and $v$ we could replace (3) with

$$f(U \cup \{s\}, s) = \begin{cases} \max_{u \in U} d(s, u) & \text{if } W(U) \leq W \text{ and } d_H(u, v) \leq h, \\ \infty & \text{Otherwise} \end{cases}$$

and the algorithm now exactly solves the weighted $k$-center problem in which each slice is constrained to have weight at most $W$ and no node can be more than $h$ edges from a center. The running time remains the same because $f(U, s)$ can still be evaluated in $O(|U|)$ time.

Example 3: Minmax range partitioning
Motivated by obtaining balanced solutions [27] discusses partitioning using range criteria. In this problem the $k$ sinks $S$ are specified in the input. For every $u \in V_{in} \setminus S$ and $s \in S$, $c_{is}$ is a given cost of servicing $u$ with sink $s$.

The range-cost of $U$ serviced by $s$ is

$$f(U, s) = \max_{u, v \in U} |c_{us} - c_{vs}|,$$
i.e., the difference between the maximum and minimum service costs. The problem is to do centered \( k \)-partitioning of the tree so as to minimize the maximum range-cost of a subtree.

[27] gives an \( O(k^2n^2) \) algorithm for this problem. While our algorithm can not solve this exact problem it can solve the variation when the range-costs are restricted to slices. That is when \( s \not\in U \) but \( s \in \Gamma(U) \) set

\[
f(U \cup \{s\}, s) = \max_{u,v \in U} |c_{us} - c_{vs}|
\]

Note that this yields a relaxed minmax monotone function. Since the range-cost can be calculated in \( O(|U|) \) time, Theorem 4 yields an \( O\left(\max(k \log k, \log n) k^2 n \log^2 n\right) \) algorithm for finding a \( k \)-partition in which the max range-cost of a slice is minimized. This improves on the original algorithm when \( k \ll n \).

We end by noting that the algorithm would remain valid if the range-cost was defined by minimizing the ratio between servicing costs within a slice rather than the absolute difference, i.e., setting

\[
f(U \cup \{s\}, s) = \max_{u,v \in U} \frac{c_{us}}{c_{vs}}
\]

### 3 Useful Properties of the Discrete Bounded-Cost Problem

This section derives the structural properties that will permit designing an algorithm. In both this section and Sect. 4, \( k \) and \( T \) are fixed given values.

**Definition 4** Let \( V \subseteq V_{in} \) and \( F \) the forest induced by \( V \).

A feasible sink configuration of \( V \) is a sink configuration \( (S, \mathcal{P}) \) of \( F \) satisfying \( |S| \leq k \) and \( f(\mathcal{P}, S) \leq T \).

Let \( S \subseteq U \subseteq V \). We say that \( U \) is served by \( S \), if there exists some \( \mathcal{P} \in \Lambda[S, U] \), such that \( (S, \mathcal{P}) \) is a feasible sink configuration of \( U \).

A feasible sink configuration (without the \( V \) stated), will denote a feasible sink configuration \( (S, \mathcal{P}) \) of \( V_{in} \).

**Definition 5** See Fig. 5a. A partial sink configuration is a feasible sink configuration \( (S, \mathcal{P}) \) of some \( V' \subseteq V_{in} \) such that \( V_{in} \setminus V' \) is a tree.

Tree \( T = (V, E) \) is a working tree for the partial sink configuration if \( S_T = V \cap V' \subseteq S \) and \( S_T \) are leaves of \( T \).

The intuition behind these definitions (developed later in the greedy construction) is that nodes in \( V' \) will be served by \( S \) while nodes in \( V \) will only be served by \( S_T \) and, possibly, by some further sinks added in \( V \).

**Definition 6** . Let \( v \in V_{in} \) and \( u \in U \subseteq V_{in} \).

\( \Pi(u, v) \) denotes the unique directed path from \( u \) to \( v \) inclusive of \( u, v \).

\( \Pi(U, v) \) denotes \( \Pi(u, v) \) where \( u \) is the closest point in \( U \) to \( v \).
Fig. 5  
(a) Initial Configuration: 
\[ S_{\text{out}} = \{s_1, s_2, s_3, s_4\}, S_T = \{s_3, s_4\} \]

(b) After Open Commit: 
\[ S_{\text{out}} = \{s_1, s_2, s_3, s_4, s_5\}, S_T = \{s_3, s_4, s_5\} \]

(c) After Two Closed Commits: 
\[ S_{\text{out}} = \{s_1, s_2, s_3, s_4, s_5\}, S_T = \{s_5\} \]

3.1 Greedy Construction

The algorithm will greedily grow a partial sink configuration \((S_{\text{out}}, P_{\text{out}})\), maintaining the property that, if a feasible sink configuration exists, \((S_{\text{out}}, P_{\text{out}})\) can always be extended to be one.

If the algorithm ever encounters \(|S_{\text{out}}| > k\), no feasible sink configuration exists. Otherwise, the algorithms terminates with \((S_{\text{out}}, P_{\text{out}})\) being a feasible sink configuration.

The algorithm also maintains an associated Working Tree \(T = (V, E)\), containing uncommitted vertices, and a set \(S_T = S_{\text{out}} \cap V\), of sinks that are still permitted to service more nodes. By the definition of working tree, sinks \(S_T\) are restricted to be leaves of \(T\).

At the start of the algorithm, \(T = T_{\text{in}}\) and \((S_{\text{out}}, P_{\text{out}}) = (\emptyset, \emptyset)\).
At each step, the algorithm will commit a subtree block $P_{\text{new}} \subseteq V$ of previously unserviced nodes to a sink $s$. This will be denoted as $\text{Commit}(P_{\text{new}}, s)$. There will be two types of commits, (Fig. 5) with the following properties:

- **Open commit**: of $P_{\text{new}} \subseteq V$ to **new** sink $s \in V \setminus S_T$.
  
  - $s$ will be added to $S_{\text{out}}, S_T$.
  
  - $P_s = P_{\text{new}}$ will be added to $P_{\text{out}}$.
  
  - $P_{\text{new}} \setminus \{s\}$ is removed from working tree $T$ which remains a tree.
  
  - $s$ becomes a leaf of $T$.

- **Closed commit**: of $P_{\text{new}} \subseteq V$ to **existing** $s \in S_T$.
  
  - If $P_{\text{new}} \neq \emptyset$, $P_{\text{new}}$ contains parent of $s$ in $V$.
  
  - $P_{\text{new}}$ is merged into $P_s$ and $s$ will be closed; no new blocks will henceforth be added to $P_s$.
  
  - $P_{\text{new}} \cup \{s\}$ is removed from $T$, which will remain a tree.

Later subsections will define the Peaking (Sect. 3.2) and Reaching (Sect. 3.4) subroutines that, respectively, implement Open and Closed commits.

The final algorithm works by requesting open and closed commits, always maintaining that $(S_{\text{out}}, P_{\text{out}})$ is extendible to some feasible sink configuration.

**Definition 7 (Extendible configurations)** Let $(S_{\text{out}}, P_{\text{out}})$ be a partial sink configuration and $T = (V, E)$ an associated working tree. Set $S_T = S_{\text{out}} \cap V$. Furthermore let $(S^*, P^*)$ be a feasible sink configuration of $V_{\text{in}}$.

1. $(S_{\text{out}}, P_{\text{out}})$ is extendible relative to $T$ and $(S^*, P^*)$ if it satisfies

   (C1) $S_{\text{out}} \subseteq S^*$ and $S^* \setminus V = S_{\text{out}} \setminus V$

   (C2) Let $P_s^*$ be the partition block in $P^*$ associated with $s \in S^*$. Then
   i. If $s \in S^* \setminus V$, then $P_s = P_s^*$ and $P_s^* \cap V = \emptyset$
   ii. If $s \in S_T$, then $P_s \cap V = \{s\}, P_s \subseteq P_s^* \text{ and } P_s^* \setminus P_s \subseteq V \setminus \{s\}$
   iii. If $s \in (S^* \setminus V) \setminus S_T$, then $P_s^* \subseteq V \setminus S_T$.

2. $(S_{\text{out}}, P_{\text{out}})$ is extendible relative to $T$ if there exists some feasible sink configuration $(S^*, P^*)$ such that $(S_{\text{out}}, P_{\text{out}})$ is extendible relative to $T$ and $(S^*, P^*)$.

The definitions below assume $(S_{\text{out}}, P_{\text{out}})$ and $T$ are fixed and given.

**Definition 8 (Self-sufficiency)** Figure 6.

Subtree $T' = (V', E')$ of $T$ is self-sufficient if $V'$ can be served by $S_{T'} = S_{\text{out}} \cap V'$. A partition of $T'$ induced by its self-sufficiency is a $P' \in \Lambda[S_{T'}, V']$, such that $(P', S_{T'})$ is a feasible sink configuration of $V'$.

Intuitively, if a subtree is self-sufficient and only connected to the remainder of the tree by one edge, it will not have any further sinks added to it.

**Definition 9 ($T_{-v}(u)$)** Figure 7.

Let $(u, v)$ be an edge of working tree $T = (V, E)$. Removing $v$ from $T$ creates a forest $\mathcal{F}_{-v}$ of disjoint subtrees of $T$.

$T_{-v}(u) = (V_{-v}(u), E_{-v}(u))$ denotes the unique subtree $T' = (V', E') \in \mathcal{F}_{-v}$ such that $u \in V'$.
Fig. 6 Self-sufficiency. 
$T' = (V', E') = T(v)$ is the tree “below” $v$. It contains sinks $S_{T'} = \{s_1, s_2, s_3\}$. If for $i = 1, 2, 3$, $f(P_i, s_i) \leq T$, then $T'$ is self-sufficient and $P_1, P_2, P_3$ is a partition of $T'$ induced by its self-sufficiency.

Fig. 7 Peaking criterion. Note that $V$ is partitioned into $V_u = V_{-u}(v)$ and $V_v = V_{-v}(u)$. $V_u$ originally contains no sinks while $V_v$ does (the black nodes). If $f(V_u, u) \leq T$, then $u$ can serve $V_u$, so no sink is needed below $u$; furthermore, if $f(V_u \cup \{v\}, v) > T$, then no node in $V_v$ can singlehandedly support $V_u$. This pinpoints the position of exactly one sink in $T_u$, to be placed at $u$.

The removal of edge $(u, v)$ splits $T$ into $T_{-v}(u)$ and $T_{-v}(v)$. The blocks committed by the algorithm will all be self-sufficient subtrees in these forms.

3.2 Subroutine: Peaking Criterion

The definition and lemmas below will justify a mechanism for greedily performing open commits. $T = (V, E)$ will always be the current working tree.

**Definition 10** (Peaking criterion) The ordered pair of points $(u, v)$ satisfies the peaking criterion if and only if (Fig. 7)

- $(u, v) \in E$,
- $V_u = V_{-v}(u)$ contains no sink in $S_T$, and
- $f(V_u, u) \leq T$ but $f(V_u \cup \{v\}, v) > T$, where $V_u = V_{-v}(u)$. 

\textcopyright Springer
The intuition is that if \((u, v)\) satisfies the peaking criterion, \(V_u\) must contain some sink \(s\) in the final feasible sink configuration and, furthermore, that we can assume that \(s = u\). If not, we can find another feasible sink configuration in which \(s = u\). This intuition is formalized in the next lemma.

**Lemma 1** (Peaking Lemma) Let \((S_{\text{out}}, P_{\text{out}})\) be extendible relative to \(T\). If \((u, v)\) satisfies the peaking criterion, then open committing \(V_u = V_{-v}(u)\) to sink \(u\) maintains \((S_{\text{out}}, P_{\text{out}})\) as extendible relative to \(T\).

**Proof** By assumption, \((S_{\text{out}}, P_{\text{out}})\) can be completed to some feasible sink configuration \((S^*, P^*)\) while satisfying (C1) and (C2) from Definition 7.

The open commit creates \((\tilde{S}_{\text{out}}, \tilde{P}_{\text{out}})\) defined by

\[
\tilde{S}_{\text{out}} = S_{\text{out}} \cup \{u\}, \quad \tilde{P}_{\text{out}} = P_{\text{out}} \cup \{\tilde{P}_u\} \text{ where } \tilde{P}_u = V_u,
\]

and \(\tilde{T} = (\tilde{V}, \tilde{E})\) where \(\tilde{V} = V_v \cup \{u\}\). By construction, \(S_{\tilde{T}} = S_T \cup \{u\}\).

Now define

\[
\tilde{S}^* = (S^* \setminus V_u) \cup \{u\}, \quad \tilde{P}_s^* = \begin{cases} P_s^* & \text{if } s \in S^* \setminus V \\ P_s^* \setminus V_u & \text{if } s \in S^* \cap V \\ V_u \cup \bigcup_{s' \in S^* \cap V_u} (P_{s'}^* \cap V_v) & \text{if } s = u \end{cases}
\]

We claim that \((\tilde{S}_{\text{out}}, \tilde{P}_{\text{out}})\) is extendible relative to working tree \(\tilde{T}\) and feasible sink configuration \((\tilde{S}^*, \tilde{P}^*)\). This is shown in five steps:

(A) \((\tilde{S}^*, \tilde{P}^*)\) is a feasible sink configuration:

Recall that \((S^*, P^*)\) is a feasible sink configuration (with \(|S^*| \leq k\)). Because \((u, v)\) satisfies the peaking condition, \(|S^* \cap V_u| \geq 1\). From this, and the definition of feasible sink configurations, it is easy to see that \(|\tilde{S}^*| \leq k\), \(\bigcup_{s \in \tilde{S}^*} P_s^* = V_{\text{in}}\), and \(\tilde{P}_{s_1}^* \cap \tilde{P}_{s_2}^* = \emptyset\) if \(s_1 \neq u, s_2 \neq u\) and \(s_1 \neq s_2\).

It only remains to confirm that \(\tilde{P}_u^* \cap \tilde{P}_s^* = \emptyset\) for \(s \neq u\).

To verify this, first note that \(\tilde{P}_u^* \subseteq V\), and recall that \(S_{\text{out}}, P_{\text{out}}\) is extendible relative to \(T\) and \((S^*, P^*)\). There are now two cases.

- \(s \in S^* \setminus V\) : then from case (C2)(iii), \(P_s^* \cap V = \emptyset\) and \(\tilde{P}_s^* = P_s^*\). Since \(\tilde{P}_u^* \subseteq V\), \(\tilde{P}_u^* \cap \tilde{P}_s^* = \emptyset\).

- \(s \in S^* \cap V_v\) : then \(\tilde{P}_s^* = P_s^* \setminus V_u\) and thus \(\tilde{P}_s^* \cap V_u = \emptyset\).

Furthermore, for any \(s' \neq s\), \(P_{s'}^* \cap P_{s^*} = \emptyset\), so

\[
\tilde{P}_s^* \cap (P_{s'}^* \cap V_v) = (P_{s'}^* \setminus V_u) \cap (P_{s'}^* \cap V_v) = \emptyset.
\]

Thus, since \(s \notin V_u\),

\[
\tilde{P}_s^* \cap P_u^* \subseteq (\tilde{P}_s^* \cap V_u) \cup \bigcup_{s' \in S^* \cap V_u} (\tilde{P}_s^* \cap P_{s'}^* \cap V_v) = \emptyset.
\]
(B) \((\tilde{S}_{\text{out}}, \tilde{P}_{\text{out}})\) is a partial sink configuration: Directly from the construction.

(C) \(\tilde{T}\) is a working tree for \((\tilde{S}_{\text{out}}, \tilde{P}_{\text{out}})\): Directly from the construction.

(D) \((\tilde{S}_{\text{out}}, \tilde{P}_{\text{out}})\) is extendible relative to \(\tilde{T}\) and \((\bar{S}^*, \bar{P}^*)\).
We must show that the conditions of Definition 7 hold. We again use the fact that those conditions held for \((S_{\text{out}}, P_{\text{out}}), T\) and \((S^*, P^*)\).

First note that by construction, \(\tilde{S}_{\text{out}} \subseteq \bar{S}^*\). We also note that

\[
(S_{\text{out}} \setminus V) = (S^* \setminus V) = (\bar{S}^* \setminus \tilde{V}) = (\tilde{S}_{\text{out}} \setminus \tilde{V})
\]

where the first equality comes from (C1) on \((S^*, P^*)\) and the second and third from the definition of \(\bar{S}^*, \tilde{S}_{\text{out}}\) and \(\tilde{V}\). Thus (C1) holds.

For (C2), we work through the sinks in \(\bar{S}^*\) one by one, showing that each satisfies its corresponding (C2) condition.

(a) \(s \in (\bar{S}^* \setminus \tilde{V}) = (S^* \setminus V)\): This is case (C2)(i) and it is satisfied because \(\bar{P}_s = P_s = P_s^* = \bar{P}_s^*\) and \(P_s^* \cap \tilde{V} \subseteq P_s^* \cap V = \emptyset\).

(b) \(s = u\): Then \(s \in S_{\tilde{T}}\) and this is case (C2)(ii). It is satisfied directly from the construction.

(c) \(s \in (S_{\tilde{T}} \setminus \{u\}) = S_T \subseteq (S^* \cap \nu) = (\bar{S}^* \cap \tilde{V})\): This is again case (C2)(ii). We are given that

\[
P_s \cap V = \{s\}, \quad P_s \subseteq P_s^*, \quad (P_s^* \setminus P_s) \subseteq (V \setminus \{s\}).
\]

Since \(\tilde{V} = \nu \cup \{u\}, \bar{P}_s = P_s\) and \(\bar{P}_s^* = P_s^* \setminus \nu\) we find easily find that

\[
\bar{P}_s \cap \tilde{V} = \{s\}, \quad \bar{P}_s \subseteq \bar{P}_s^*, \quad (\bar{P}_s^* \setminus \bar{P}_s) \subseteq (\tilde{V} \setminus \{s\}).
\]

So, (C2)(ii) holds.

(d) \(s \in (\bar{S}^* \cap \tilde{V}) \setminus S_{\tilde{T}} = (S^* \cap V) \setminus S_T\): Note this implies \(s \in (\tilde{V} \setminus S_{\tilde{T}}) \subseteq \nu\).

This is (C2)(iii) for both \((S^*, P^*)\) and \((\bar{S}^*, \bar{P}^*)\). Now note

\[
\bar{P}_s^* = (P_s^* \setminus \nu) \subseteq ((V \setminus S_T) \setminus \nu) = (V \setminus S_T) = (\tilde{V} \setminus S_{\tilde{T}})
\]

where the first inclusion comes from (C2)(iii) on \(s \in S^*\) giving \(P_s^* \subseteq V \setminus S_T\). Thus \(\bar{P}_s^* \subseteq \tilde{V} \setminus S_{\tilde{T}}\) and (C2)(iii) is satisfied.

\[\square\]

The algorithm keeps attempting to add sinks by finding edges that satisfy the peaking criterion. The next lemma, with its corollary, exactly characterizes when no such edge exists and the process must stop. They follow immediately from Definitions 1 and 10.

**Lemma 2** Suppose for some \(u, v\), \(f(V_{\nu} \cup \{v\}) \cup \{v\}, V) > T\), and \(S \cap V_{\nu}(u) = \emptyset\). Then there exists \(u', v' \in V_{\nu} \cup \{v\}\) satisfying the peaking criterion.

**Corollary 1** Let \((S_{\text{out}}, P_{\text{out}})\) be an extendible partial sink configuration relative to working tree \(T\). If no directed pair \((u, v)\) satisfy the peaking criterion, then exactly one of the following four situations occurs:
1. \( S_T = \emptyset \): Then for all \( s \in V \), \( f(V, s) \leq T \). Furthermore, for all \( s \in V \), Commit \((V, s)\) will create a feasible sink configuration \((\bar{S}_{out}, \bar{P}_{out})\).

2. \( S_T = \{s\} \) for some \( s \in V \): Then \( f(V, s) \leq T \). Furthermore, Commit \((V, s)\) will create a feasible sink configuration \((\bar{S}_{out}, \bar{P}_{out})\).

3. \(|S_T| = |V| = 2\) Let \( V = S_T = \{s, s'\} \). Then Commit \((\{s\}, s)\) followed by Commit \((\{s'\}, s')\) will create a feasible sink configuration \((\bar{S}_{out}, \bar{P}_{out})\).

4. \(|S_T| \geq 2\) and \(|V| > 2\). Our algorithm will repeatedly place sinks using the Peaking Lemma until no edge satisfying the peaking condition can be found. It maintains the invariant that \((\bar{S}_{out}, \bar{P}_{out})\) remains extendible relative to working tree \(T\).

Corollary 1 implies that if some feasible sink configuration exists, no edge satisfying the lemma can be found and \(|S_T| < 2\), or \(|S_T| = |V| = 2\), then the resulting sink configuration constructed is feasible.

### 3.3 The Hub Tree

The peaking lemma will be the only method of adding new sinks to \(S_{out}\). If the peaking criterion no longer hold for any pair \(u, v \in V\), but \(P_{out}\) still does not contain a full partition, another mechanism will be needed to perform closed commits of unserviced blocks to already existing sinks.

Section 3.4 introduces the Reaching Criterion for this. This will first require defining the Hub Tree.

In what follows, \((\bar{S}_{out}, \bar{P}_{out})\) and working tree \(T = (V, E)\) are fixed. \(S_T = S_{out} \cap V\).

#### Definition 11 (Hubs)

See Fig. 8. Recall that nodes in \(S_T\) are leaves of \(T\). Assume \(|S_T| \geq 2\), \(|V| > 2\) and that \(T\) is rooted at some non-sink \(r\) such that at least two of \(r\)’s children are sinks or have sink descendants

- Let \(H(S_T) \subseteq V\) be the set of lowest common ancestors of all pairs of sinks in \(T\). The nodes in \(H(S_T)\) are the hubs associated with \(S_T\).
- The hub tree \(T_{H(S_T)} = (V_{H(S_T)}, E_{H(S_T)})\) is the rooted minimal subtree of \(T\) that contains all vertices and edges contained in all of the paths \(\Pi(s, r)\) where \(s \in S_T\).
- For \(u \in V_{H(S_T)}\), set \(T(u) = (V(u), E(u))\) to be the subtree of \(T\) rooted (down) at \(u\).

#### Definition 12 (Outstanding branches)

A node \(w \in V_{H(S_T)}\) branches out to \(\eta\) if \(\eta\) is a neighbor of \(w\) in \(T\) that does not exist in \(V_{H(S_T)}\). The subtree \(T' := T_{-w}(\eta)\) is called an outstanding branch; we say that \(T'\) is attached to \(w\).

From the definition of the hub tree, outstanding branches contain no sinks.

#### Definition 13 (Bulk path)

Given \(u, v \in V_{H(S_T)}\), the bulk path \(BP(u, v)\) is the union of path \(\Pi(u, v)\) with all the nodes in all outstanding branches that are attached to any...
node in $\Pi(u, v)$. $BP(v, v)$ denotes the special case of the union of $v$ and of all the outstanding branches falling off of $v$.

We now describe what occurs when the peaking criterion is inapplicable.

**Definition 14** (RC-viable) $T$ is RC-viable (with respect to $S_T$) if, for every $T' = (V', E')$ that is an outstanding branch attached to $w \in V_H(S_T)$, $f(V' \cup \{w\}, w) \leq T$.

**Lemma 3** If no ordered pair $(u, v)$ satisfies the peaking criterion then $T$ is RC-viable (with respect to $S_T$).

**Proof** Let $T' = (V', E') = T_{-w}(\eta)$ be an arbitrary outstanding branch attached to some node $w \in V_H(S_T)$. If $f(V' \cup \{w\}, w) > T$ then from Lemma 2, $T'$ would contain an ordered pair $(u, v)$ satisfying the peaking criterion. Since no such pair exists, $f(V' \cup \{w\}, w) \leq T$. This is true for all outstanding branches and thus $T$ is RC-viable. 

**Lemma 4** Let $(S_{\text{out}}, P_{\text{out}})$ be extendible relative to working tree $T = (V, E)$ and feasible sink configuration $(S^*, P^*)$ and suppose that $T$ is RC-viable.

Then there exists a feasible sink configuration $(\tilde{S}^*, \tilde{P}^*)$ such that $(S_{\text{out}}, P_{\text{out}})$ is feasible relative to $T$ and $(\tilde{S}^*, \tilde{P}^*)$, that has the further property that none of the sinks in $\tilde{S}^*$ are located in any of the outstanding branches of $T$.

**Proof** Suppose $S^*$ did contain a sink $s$ located in an outstanding branch of $T$ attached to some $w \in V_H(S_T)$. Then RC-viability implies that $s$ could be moved to $w$ (removing $s$ if $w$ was already in $S^*$ while maintaining the feasibility of the sink configuration (this might require modifying $P^*$) and the validity of (C1) and (C2). 

This last lemma permits assuming that all sinks in $S^* \setminus S_{\text{out}}$ are in $V_H(S)$. 

---

*Fig. 8 Visualization of hub tree $T_H(S_T)$ with root $r$. Areas enclosed by dashed lines are outstanding branches (and are not in the hub tree), filled circles denote sinks, and unfilled circles denote hubs. $BP(u, w)$ is the union of the path $\Pi(u, w)$ in the hub tree and the two outstanding branches $T'_1$ and $T'_2$ along with the edges connecting $T'_1$, $T'_2$ to the path. $T(w)$ is the tree rooted at $w$ and includes everything below it, including $T'_1$, $T'_2$ and sinks $s_1$-$s_4$.**
3.4 Subroutine: Reaching Criterion

The definition and lemmas below justify a mechanism for greedily performing closed commits. Note that, unlike in the open-commit case, the process might require performing multiple closed commits simultaneously.

**Definition 15** *(Reaching criterion)* Let \( T \) be RC-viable with respect to \( S \) and \( (u, v) \in E \) with \( u, v \in V_{H(S_T)} \). Then \( (u, v) \) satisfies the Reaching Criterion (RC) if and only if (Fig. 9)

- \( v \) is the parent of \( u \) (when \( T \) is rooted at \( r \)).
- \( T_{-v}(u) \) is self-sufficient and
- \( BP(v, v) \cup V_{-v}(u) \) is not self-sufficient.

The intuition (See Fig. 9) is that \( BP(v, v) \cup V_{-v}(u) \) should not be served by any sink \( s \in S_T \cap V_{-v}(u) \). If it was, then because \( BP(v, v) \cup V_{-v}(u) \) is not self-sufficient, \( V_{-v}(u) \) must contain at least one more sink \( s \notin S_T \). But, in that case, it would be more “efficient” to place \( s' \) on \( v \) and not in \( T_{-v}(u) \). Thus, if \( (u, v) \) satisfies the reaching criterion, it makes sense to peel off \( T_{-v}(u) \) and its sinks from the working tree. This is formalized in the next lemma.

**Lemma 5** *(Reaching Lemma)* Let \( (S_{out}, P_{out}) \) be extendible relative to working tree \( T \).

Suppose \( T \) is RC-viable with respect to \( S_T \) and \( (u, v) \) is a directed edge in \( T \) that satisfies the reaching criterion. Set \( S_T(u) = V_{-v}(u) \cap S_T \).

Partition \( T_{-v}(u) \) into the subtrees induced by the corresponding sinks in \( S_T(u) \) as implied by the self-sufficiency of \( T_{-v}(u) \) (Definition 8).

Then committing those subtrees to the blocks associated with those sinks using the closed commits on \( S_T(u) \) as defined in Sect. 3.1 (Fig. 5c), maintains \( (S_{out}, P_{out}) \) as extendible relative to \( T \).
Proof. Since \((S_{\text{out}}, \mathcal{P}_{\text{out}})\) is extendible relative to \(T\), it can be completed to some feasible sink configuration \((S^*, \mathcal{P}^*)\) while satisfying (C1) and (C2).

As in the proof of Lemma 1, write \(V_u = V_{-u}(u)\) and \(V_v = V_{-u}(v) = V \setminus V_u\).

Set \((\bar{S}_{\text{out}}, \bar{P}_{\text{out}})\) and \(\bar{T}\) to be the corresponding values after the closed commits. Note that \(\bar{S}_{\text{out}} = S_{\text{out}}\) and \(\bar{V} = V_v\). \(S_{\bar{T}} = S_{\bar{T}} \cap V_v\).

Define \(P_{\text{new}}(s)\) be the nodes in \(V_u\) that are served by \(s \in S_{\bar{T}}(u)\) as implied by the self-sufficiency of \(T_{-v}(u)\). Then \(\bigcup_{s \in S_{\bar{T}}(u)} P_{\text{new}}(s) = V_u\) and

\[
\bar{P}_s := \begin{cases} 
  P_s = P^* & \text{if } s \in S_{\text{out}} \setminus V_u, \\
  P_s \cup P_{\text{new}}(s) = (P^*_s \setminus V) \cup P_{\text{new}}(s) & \text{if } s \in S_{\text{out}} \cap V_u = S_{\bar{T}}(u).
\end{cases}
\]

We will show that \((\bar{S}_{\text{out}}, \bar{P}_{\text{out}})\) is extendible relative to \(\bar{T}\) and can be completed to some feasible sink configuration \((\bar{S}^*, \bar{P}^*)\).

The proof will follow by proving the same (A)-(D) steps as in the proof of Lemma 1, albeit in a different order. The proofs of (B) and (C) below follow directly from the construction.

(B) \((\bar{S}_{\text{out}}, \bar{P}_{\text{out}})\) is a partial sink configuration:

(C) \(\bar{T}\) is a working tree for for \((\bar{S}_{\text{out}}, \bar{P}_{\text{out}})\):

We now need to define \((\bar{S}^*, \bar{P}^*)\) and prove

(A) \((\bar{S}^*, \bar{P}^*)\) is a feasible sink configuration:

Set \(S^*_u = (S^* \cap V_u) \setminus S_{\bar{T}}(u)\). These are the sinks in \(S^* \cap V_u\) that are not going to be used anymore. Set \(s(v)\) to denote the unique sink such that \(v \in P^*_s(v)\). Note that \(s(v) \in V\).

The definition of \((\bar{S}^*, \bar{P}^*)\) will depend upon the location of \(s(v)\).

Variant 1: If \(s(v) \in V_{-u}\) Set \(\bar{S}^* = (S^* \setminus S^*_u) \cup \{v\}\) and,

\[
\forall s \in \bar{S}^*, \text{ set } \bar{P}^*_s := \begin{cases} 
  (P^*_s \setminus V) \cup P_{\text{new}}(s) & \text{if } s \in S_{\bar{T}}(u) \\
  P^*_s(\bar{v}) \cap \bar{V} & \text{if } s = v \\
  P^*_s & \text{Otherwise}
\end{cases}.
\]

Variant 2: If \(s(v) \in V_{-u}\) Set \(\bar{S}^* = (S^* \setminus S^*_u)\) and,

\[
\forall s \in \bar{S}^*, \text{ set } \bar{P}^*_s := \begin{cases} 
  (P^*_s \setminus V) \cup P_{\text{new}}(s) & \text{if } s \in S_{\bar{T}}(u) \\
  P^*_s(\bar{v}) \cap \bar{V} & \text{if } s = s(v) \\
  P^*_s & \text{Otherwise}
\end{cases}.
\]

For both variants it is straightforward to see that \(\bigcup_{s \in \bar{S}^*} \bar{P}^*_s = V_{\text{in}}\), the \(\bar{P}^*_s\) are disjoint and, for all \(s \in \bar{S}^*, f(\bar{P}^*_s, s) \leq \bar{T}\).

To prove (A) it only remains to show that \(|\bar{S}^*| \leq k\). For variant 2, this is immediate, since \(|\bar{S}^*| = |S^*| - |S^*_u| \leq |S^*| \leq k\).

For variant 1, the important observation is that since \(s(v) \in S^* \cap V_u\), \(BP(v, v) \cup V_u\) is not serviced by a sink in \(S^* \cap V_v\). The non self-sufficiency of \(BP(v, u) \cup V_u\) then
implies that $S^* \cap V_u$ must include some sink $s' \in S^*_u$, so $|S^*_u| \geq 1$. Thus $|\tilde{S}^*| = |S^*| + 1 - |S^*_u| \leq k + 1 - |S^*_u| \leq k$ as well.

It now only remains to prove that

$$(\text{D}) \, (\tilde{S}_{\text{out}}, \tilde{P}_{\text{out}}) \text{ is extendible relative to } \tilde{T} \text{ and } (\tilde{S}^*, \tilde{P}^*)$$

We must show that the conditions of Definition 7 hold. We use the fact that those conditions held for $(S_{\text{out}}, P_{\text{out}})$, $T$ and $(S^*, P^*)$.

The proof closely follows the structure of the proof of (D) in Lemma 1.

First note that, by construction, $\tilde{S}_{\text{out}} \subseteq \tilde{S}^*$. We also note that

$$(\tilde{S}_{\text{out}} \setminus \tilde{V}) = (S_{\text{out}} \setminus V) \cup S_T(u) = (S^* \setminus V) \cup S_T(u) = (\tilde{S}^* \setminus \tilde{V})$$

where the second equality comes from (C1) on $(S^*, P^*)$ and the first and third from the definition of $\tilde{S}^*$ and $\tilde{S}_{\text{out}}$. Thus (C1) holds.

For (C2), we work through the sinks in $\tilde{S}$ one by one, showing that each satisfies its corresponding (C2) condition. We note that the analysis uses $V_u$ and $\tilde{V}$ interchangeably.

(a) $s \in (\tilde{S}^* \setminus \tilde{V}) = (S^* \setminus V)$. This is case (C2)(i) and it is satisfied because $\tilde{P}_s = P_s = P^*_s = \tilde{P}^*_s$ and $\tilde{P}^*_s \cap \tilde{V} \subseteq P^*_s \cap V = \emptyset$.

(b) $s \in S_T(u)$. This is again case (C2)(i). By construction $\tilde{P}^*_s \cap \tilde{V} = \emptyset$ and

$$\tilde{P}_s = (P^*_s \setminus V) \cup P_{\text{new}}(s) = \tilde{P}^*_s$$

(c) $s \in S_T = S_T \cap V_u$ and $s \neq s(v)$. This is case (C2)(ii).

We are given that

$$P_s \cap V = \{s\}, \quad P_s \subseteq P^*_s, \quad P^*_s \setminus P_s \subseteq V \setminus \{s\}.$$

Furthermore, because $s \neq s(v)$, $P^*_s \cap V \subset \tilde{V}$ so $P^*_s \setminus P_s \subseteq \tilde{V} \setminus \{s\}$.

Since $s \in P_s$, $\tilde{P}_s = P_s$ and $\tilde{P}^*_s = P^*_s$ we easily find that

$$\tilde{P}_s \cap \tilde{V} = \{s\}, \quad \tilde{P}_s \subseteq \tilde{P}^*_s, \quad \tilde{P}^*_s \setminus \tilde{P}_s \subseteq \tilde{V} \setminus \{s\}.$$

So, (C2)(ii) holds.

(d) $s = v$ in variant 1. Then $s \in (\tilde{S}^* \setminus \tilde{V}) \setminus S_T$, so this is case (C2)(iii).

By definition, $\tilde{P}^*_s = P^*_s \cap V_v \subseteq \tilde{V}$. Since $\tilde{P}^*_s \cap \tilde{S}^* = s(v)$ and $s(v) \in V_u$, this implies $\tilde{P}^*_v \subseteq \tilde{V} \setminus S_T$. So (C2)(iii) holds.

(e) $s \in (\tilde{S}^* \cap \tilde{V}) \setminus S_T = (\tilde{S}^* \cap \tilde{V}) \setminus (S_T \subseteq (S^* \cap V) \setminus S_T$.

This is case (C2)(iii) for both $(S^*, P^*)$ and $(\tilde{S}^*, \tilde{P}^*)$.

From (C2)(iii) on $(S^*, P^*)$ we have $P^*_s \subseteq \tilde{V} \setminus S_T$.

In the case of variant 1, because $s(v) \in V_u$, $s \neq s(v)$ and thus $\tilde{P}^*_s = P^*_s \subseteq \tilde{V}$.

In the case of variant 2, if $s = s(v)$, then $\tilde{P}^*_s = P^*_s \setminus V_u \subseteq V_u \subseteq \tilde{V}$.

If $s \neq s(v)$, then $\tilde{P}^*_s = P^*_s \subseteq V_v = \tilde{V}$.

So, in all cases we have $\tilde{P}^*_s \subseteq P^*_s$, $P^*_s \subseteq V \setminus S_T$ and $\tilde{P}_s \subseteq \tilde{V}$.

Thus, for all such $s$, $\tilde{P}^*_s \subseteq (\tilde{V} \setminus S_T) \subseteq (\tilde{V} \setminus S_T)$ and (C2)(iii) holds.

$\square$
It is important to note that after $T_{-v}(u)$ is removed by the reaching criterion, the remaining tree $T$ might no longer be RC-viable. The peaking criterion would need to be checked again on $T$, in order to reimpose RC-viability.

### 3.4.1 Testing for Self-sufficiency

Self-sufficiency is expensive to test. The following specialization will be more efficient for algorithmic purposes. In what follows, we assume the hub tree has been created and is rooted at some non-sink vertex $r$.

**Definition 16 (Recursive self-sufficiency)** Let $v \in V_{H(S_T)}$.

Subtree $T(u)$ is recursively self-sufficient if for all $u \in V_{H(S_T)} \cap V(v)$, $T(u)$ is self-sufficient.

Recursive self-sufficiency can be tested in a bottom-up manner.

**Lemma 6** Let $T$ be RC-viable and $v \in V_{H(S_T)}$.

1. Let $u \in V_{H(S_T)}$ be a child of $v$ such that
   
   (i) $T(u) = T_{-v}(u)$ is recursively self-sufficient, and
   
   (ii) there is a sink $s \in S_T \cap V_{-v}(u)$ such $f(BP(v, s), s) \leq T$.

   Then $BP(v, v) \cup T_{-v}(u)$ is self-sufficient.

2. Now suppose that in addition to the existence of some $u$ that satisfies (i) and (ii) above, for every other child $u'$ of $v$ in $V_{H(S_T)}$, $T(u') = T_{-v}(u')$ is recursively self-sufficient.

   Then $T(v)$ is recursively self-sufficient.

**Proof** From (ii) we know that $BP(v, s)$ is self-sufficient. Remove $BP(v, s)$ from $T_{-v}(u)$. Now consider the remaining rooted forest induced by $V_{-v}(u) \setminus BP(v, s)$.

By the recursive self-sufficiency of $T_{-v}(u)$, each rooted tree in this forest is self-sufficient. We have thus shown that $BP(v, v) \cup T_{-v}(u)$ is served by $S_T \cap T_{-v}(u)$. (1) follows.

To prove (2), similarly note that since $T(u')$ is recursively self-sufficient for every child $u'$ of $v$ in $T_{H(S_T)}$, every node $v' \in V_{H(S_T)} \cap V(v)$ except for $v$ must satisfy that $T(v')$ is self-sufficient. It thus suffices to prove that $T(v)$ itself is self-sufficient.

From (1) it is already known that $BP(v, v) \cup V_{-v}(u)$ is self-sufficient. Removing $BP(v, v) \cup V_{-v}(u)$ from $T(v)$ leaves a rooted forest in which the root of each forest is a child $u'$ of $v$ in $T(v)$. Since each such tree $T(u')$ is given to be self-sufficient, all of $T(v)$ is self-sufficient.

Let $v$ be recursively self-sufficient. If Lemma 6 (1) (i) and (ii) hold for some $s, u$, we say that $s$ is a witness to Lemma 6 for $T(v)$. For every such $v$, store all of its witnesses at $T(v)$.

Then, if $T'$ is a recursively self-sufficient tree, from the proof of Lemma 6 (2), it is easy to retrieve in $O(|V'|)$ time a partition $P'$ of $T'$ that witnesses the self-sufficiency of $T'$. 

Springer
More explicitly, view $T'$ as a recursively self-sufficient rooted forest. If $T'$ is not empty, choose any maximal subtree $T''$ of $T'$. Let $v''$ be its root and $s''$ any witness for $T''$. Remove $BP(v'', s'')$ from $T'$ and commit all of the nodes in $BP(v'', s'')$ to $s''$.

What remains of $T'$ is a new, smaller, still recursively self-sufficient, rooted forest $\bar{T'}$. Repeat this process on $\bar{T'}$ until all nodes have been removed and committed to some sink.

Recursive self-sufficiency will provide an efficient test for the reaching criterion via the following immediate corollary to Lemma 6.

**Corollary 2** Let $T$ be RC viable. and $v \in V_{H(ST)}$. Let $u_i, i = 1, \ldots, j$ be the children in $T_{H(ST)}$ of $v$ and assume that all the $T_{-v}(u_i)$ are recursively self-sufficient. Then exactly one of the following two cases must occur with the associated consequences:

(i) $\exists i$, such that for all sinks $s \in S_T \cap T_{-v}(u_i)$, $f(BP(v, s), s) > T$.
   $\Rightarrow$ $(u_i, v)$ satisfies the reaching criterion.

(ii) $\forall i$, there exists sink $s \in S_T \cap T_{-v}(u_i)$ such that $f(BP(v, s), s) \leq T$.
   $\Rightarrow T(v)$ is recursively self-sufficient.

Note: Only the conditions of (i) and (ii) are disjoint. It is quite possible that both of their conclusions simultaneously hold, i.e., that (i) occurs and that $T(v)$ is recursively self-sufficient.

The algorithm can walk up the hub-tree from its leaves (sinks), checking recursive self-sufficiency using case (ii) of the corollary. If case (ii) ever fails, case (i) must occur, yielding a $(u_i, v)$ pair satisfying the reaching criterion. The process also terminates if it reaches $r$ and finds that $T(r) = T$ is recursively self-sufficient. But, in that case the algorithm itself terminates because $T$ can be supported by $S$. This automatically leads to the next corollary

**Corollary 3** Let $T$ be RC viable. Then at least one of the following two cases must occur

(i) $\exists (u, v) \in E_{H(ST)}$ that satisfies the reaching criterion

(ii) $T$ is recursively self-sufficient and can be fully evacuated to the nodes in $S$.

### 3.5 The Evolution of the Hub Tree

We have seen how, when the peaking criterion can no longer be applied, the working tree $T = (V, E)$ is RC-viable with respect to $S_T = V \cap S_{out}$. Let $T_{H(ST)} = (V_{H(ST)}, E_{H(ST)})$ be the directed hub-tree with root $r$.

From Corollary 3, either $T$ itself is recursively self-sufficient (and the algorithm terminates) or there exists some $(u, v)$ in $T_{H(ST)}$ that satisfies the reaching criterion. This permits removing the tree $T_{-v}(u)$ rooted at $u$, resulting in a new tree $\bar{T}$. Note that the new tree $T(v)$ in $\bar{T}$ is no longer guaranteed to be self-sufficient. Also, since $\bar{T}$ might no longer be RC-viable, it needs to be checked again for the peaking criterion.

The remainder of this subsection examines what can happen next. It will show that if $\bar{T}$ does not remain RC-viable then there is exactly one edge, lying on a very specific known path in what remains of $T_{H(ST)}$, that satisfies the peaking criterion. The removal
Fig. 10 A labeled hubtree $T_H(S_T)$. Solid nodes are the sinks in $S_T$; unfilled nodes are hubs. Triangles are outstanding branches. $p(u_1) = u_2$, $p_H(u_1) = h_7$. $\Pi(h_9, h_{13})$ is the path $h_9, u_3, u_4, u_5, h_{13}$. $(u_3, u_4)$ is on path $\Pi(h_9, h_{13})$.

Fig. 11 Illustration of Lemma 7. A labeled hubtree $T_H(S_T)$. Solid nodes are the sinks in $S$; unfilled nodes are hubs. Outstanding branches are not shown. $(u_1, v_1), (u_2, v_2)$ and $(u_3, v_3)$ with $p(u_i) = v_i$ are all known to satisfy the reaching criterion. Note that $p_H(u_1) = h_{22}, p_H(u_2) = h_{22}$ and $p_H(u_3) = h_{16}$. If $V_{-v_1}(u_1)$ is removed, then the only possible location upon which an edge $(\bar{u}, \bar{v})$ satisfying the peaking criterion can lie is on the path $\Pi(v_1, h_{22})$. Such an edge will exist if and only if $f(V_{-h_{22}}(u_1) \cup \{h_{22}\}, h_{22}) > T$ in the remaining tree. A similar situation occurs if, instead, $V_{-v_2}(u_2)$ is removed. Note that $v_2 = h_{20}$, the hub-parent of $u_2$. Because of this, no new peaking edge can exist of this edge will result in a new RC-viable $\bar{T}$. Deriving this will require the following definitions:

**Definition 17**. In what follows $u, v \in V_H(S_T)$. See Fig. 10.

- If $u \neq r$, set $p(u)$ to be the parent of $u$ (which is in $V_H(S)$).
- Set $p_H(u)$ to be lowest ancestor of $u$ on $\Pi(p(u), r)$ that is a hub-node. $p_H(u)$ is the hub-parent of $u$. 

© Springer
see Figure 10. Let $T$ be RC-viable relative to $S_T$, $T_{H(S_T)}$ be rooted at $r$ and $(u, v)$ an edge that satisfies the reaching criterion.

Now let $(\tilde{S}_{out}, \tilde{P}_{out})$, $\tilde{T}$, $S_{\tilde{T}}$ be the result after applying the Reaching Lemma to $(u, v)$, removing $T_{-v}(u)$ and (close-)committing its nodes to the appropriate sinks in $\{ s \in S_T \cap V_{-v}(u) \}$. Note that $\tilde{T}$ is $T$ with subtree $T_{-v}(u)$ removed.

**Definition 18** See Fig. 10. Let $T = (V, E)$ be a (undirected) tree with $v, h, \bar{u}, \bar{v} \in V$ and $(\bar{u}, \bar{v}) \in V$. Then directed edge $(\bar{u}, \bar{v})$ is on directed path $\Pi(v, h)$, if (i) $\bar{u}, \bar{v} \in \Pi(v, h)$ and (ii) $\bar{v} \in (\bar{u}, h)$.

We can now understand exactly where any new peaking edge can lie. There are two possible cases.

**Lemma 7** (See Fig. 11) Let $(\tilde{S}_{out}, \tilde{P}_{out})$, $\tilde{T}$, $S_{\tilde{T}}$ be as described above. Set $h = p_H(u)$. Further suppose that $h \neq r$ or that $h = r$ and $h$ has at least three children in $T_{H(S_T)}$. Finally, if $v \neq h$, set $v'$ to be the unique child of $h$ such that $v' \in \Pi(v, h)$. Then

(A) $\tilde{T}$ contains at most one directed edge $(\bar{u}, \bar{v})$ that satisfies the peaking criterion.

Furthermore, if such a $(\bar{u}, \bar{v})$ exists, then $(\bar{u}, \bar{v})$ is on path $\Pi(v, h)$.

(B) Such a $(\bar{u}, \bar{v})$ exists if and only if $h \neq v$ and $f(V_{-h}(v') \cup \{h\}, h) > \mathcal{T}$.

**Proof** It is not difficult to see that if $(\bar{u}, \bar{v})$ is not on the directed path $\Pi(v, h)$, then either $(\bar{u}, \bar{v})$ is in some outstanding branch of $T$, with $\bar{v} = p(\bar{u})$ or $T_{-v}(\bar{u})$ contains a sink in $S_{\tilde{T}}$. In both these cases, $(\bar{u}, \bar{v})$ can not satisfy the peaking criterion. Thus $(\bar{u}, \bar{v})$ must lie on $\Pi(v, h)$. Since $\Pi(v, h)$ can only contain one peaking edge, (A) follows.

To prove (B) first note that if $h = v$ then $\Pi(v, h)$ is just the point $h$ so it can not contain an edge. If $h \neq v$, then (B) follows directly from Lemma 2 and path and set monotonicity.

**Lemma 8** (See Fig. 12) Let $(\tilde{S}_{out}, \tilde{P}_{out})$, $\tilde{T}$, $S_{\tilde{T}}$ be as described above. Further suppose that $p_H(u) = r$ and $r$ had only two children in $T_{H(S_T)}$. Define $h'$ to be the only remaining hub-child of $r$ in $\tilde{T}$. Set $v'' = p(h)$.
(A) Then $\bar{T}$ contains at most one directed edge $(\bar{u}, \bar{v})$ that satisfies the peaking criterion. Furthermore, if such a $(\bar{u}, \bar{v})$ exists, then $(\bar{u}, \bar{v})$ is on path $\Pi(v, h')$.

(B) Such a $(\bar{u}, \bar{v})$ exists if and only if $f(V_{-h'}(v'') \cup \{h'\}, h') > T$.

Proof To prove (A), note that after applying the Reaching Lemma to $(u, v), \bar{V}_{-r}(v')$ no longer contains any sinks. All remaining sinks in $\bar{T}$ are in $T(h')$, (which is either a single sink or a hub node of $T_{H(S_F)}$).

Similar to the proof of the previous lemma, it is easy to see that if $(\bar{u}, \bar{v})$ is not on the directed path $\Pi(v, h')$, then either $(\bar{u}, \bar{v})$ is in some outstanding branch of $T$, or $T_{-\bar{v}}(\bar{u})$ contains a sink in $S_{\bar{T}}$. In both these cases, $(\bar{u}, \bar{v})$ can not satisfy the peaking criterion. Again, since $\Pi(v, h)$ can only contain one peaking edge, (A) follows.

To prove (B) first note that $v \neq r$ and $v$ and $h'$ are in different subtrees of $r$. Thus $\Pi(v, h')$ contains a path of length at least two in $\bar{T}$. (B) again follows directly from Lemma 2 and path and set monotonicity.

Lemmas 7 and 8 imply that after the Reaching Lemma is applied, at most one edge in the remaining working tree $T$, located on an easily identifiable path, can satisfy the peaking criterion before $T$ becomes RC-viable again.

4 Designing an Algorithm for The Discrete Bounded Cost Problem

Combining the pieces from Sect. 3 yields a generic algorithm for solving the discrete bounded-cost problem. For given $T$. This is shown in Algorithm 1.

This algorithm initializes by setting $(S_{out}, P_{out}) = (\emptyset, \emptyset), T = (V, E) = T_{in}$ and $S = S_{out} \cap V = \emptyset$. This $(S_{out}, P_{out})$ is trivially feasible relative to $T$.

The algorithm then attempts to find edges $(u, v)$ that satisfy the peaking criterion. Every time it finds such an edge it performs an open commit. From Lemma 1, this maintains $(S_{out}, P_{out})$ as being feasible relative to $T$.

If adding a sink via the peaking criterion ever finds $|S_{out}| > k$, the algorithm reports that no feasible sink configuration exists. If no edge satisfying the criterion can be found and $|S| = |S_{out} \cap V| < 2$ or $|V| = |S| = 2$ then the algorithm finds a feasible sink configuration using Corollary 1. More specifically

- if $|S| = 0$ then Commit$(V, v)$ for any $v \in V$,
- if $S = \{s\}$ then Commit$(V, s)$,
- if $V = S = \{s, s'\}$ then Commit$(\{s\}, s)$ and Commit$(\{s'\}, s')$.

In all of these cases the algorithm concludes with $(S_{out}, P_{out})$ being a feasible sink configuration for the original $T_{in}$.

If no edge satisfying the peaking criterion exists and $|S| \geq 2$ and $|V| > 2$ then $T$ is RC-viable. The algorithm then attempts to find an edge satisfying the reaching criterion. If it succeeds, it performs the corresponding closed commits and returns to trying to find an edge satisfying the peaking criterion. By Lemma 5 this maintains $(S_{out}, P_{out})$ as being feasible relative to $T$. If no edge satisfying the reaching criterion exists, then by Corollary 3, $T$ can be fully committed to the sinks in $S$; the resulting sink configuration $(S_{out}, P_{out})$ is a feasible sink configuration for the original $T_{in}$.

Springer
Algorithm 1 Generic Bounded cost algorithm for fixed $T$.  

```plaintext
1: $T := (V, E) := T_{in}$
2: $S := S_{out} = \emptyset$, $P_{out} := \emptyset$.
3:
4: procedure PEAKING.PHASE
5: while Some edge $(u, v) \in E$ satisfies Peaking Condition do  
6:   Commit($V_{\neq}(u) \cup \{v\}$, $s$)  
7: end while
8: if $S = \{s\}$ for some $s \in S$ then  
9:   Commit($V, s$).  
10: else if $S = \emptyset$ then  
11:   Choose any $v \in V$ and Commit($V, v$).  
12: else if $V = S = \{s, s'\}$ then  
13:   Commit($\{s\}, s$) and Commit($\{s'\}, s'$).  
14: end if
15: if $|S_{out}| > k$ then  
16:   BREAK and Return “Infeasible”  
17: end if
18: end procedure
19:
20: procedure REACHING.PHASE
21: if $(u, v) \in E_{H(T)}$ satisfying Reaching Condition is found then  
22:   Remove $V_{\neq}(u)$ from $T$; Commit blocks for $V_{\neq}(u)$  
23: else  
24:   Set $T = \emptyset$; Commit blocks for all of $V$  
25: end if
26: end procedure
27:
28: // Start Algorithm
29: while Tree $T$ is not empty do
30:   PEAKING.PHASE
31: if Tree $T$ is not empty then
32:   Create Hub Tree $T_{H(S)}$ from $T$, $S$
33:   REACHING.PHASE
34: end if
35: end while
36: Output $S_{out}$, $P_{out}$
```

The algorithm is generic because it does not specify an order or methods for finding edges that satisfy the peaking or reaching criteria. The next section develops efficient techniques for both. It proceeds as follows:

4.1.1 Implementation of the first peaking phase via a tree centroid decomposition
4.1.2 Implementation of all other peaking phases.
4.2 Creation of the hub tree after a peaking phase
4.3 Implementation of the reaching phase after constructing the hub tree.

The decomposition into these parts will make it easier to apply parametric searching in Sect. 5 to prove Theorem 2.
4.1 Implementing the Peaking Phase

4.1.1 The First Peaking Phase via Tree Centroid Decomposition.

The peaking phase checks the peaking criterion on all possible directed edges \((u, v)\), committing \(V_{-v}(u)\) to \(u\) if appropriate. Explicitly checking every edge would require \(O(n)\) oracle calls. This subsection develops a method that only requires \(O(\log n)\) (amortized) oracle calls plus \(O(n \log n + nk)\) extra work for the first peaking phase. Section 4.1.2 will then show how to implement each of the \(O(k)\) subsequent peaking phases using \(O(\log n)\) (actual) oracle calls per phase plus \(O(nk)\) total extra work over all the remaining phases.

We start by noting that information garnered when checking an edge for the peaking criterion will often imply that many other edges will not satisfy the criterion and therefore need not be tested. The algorithm will take advantage of this and create an order for checking the edges – based on a recursive centroid decomposition of \(T_{in}\) – that will guarantee that if many oracle calls are made, the average size of an oracle call will be small. The asymptotic subadditivity of the oracle \(A\) will then yield an amortized running time of the first peaking stage equivalent to \(O(\log n)\) oracle calls.

**Definition 19** Let \((u, v)\) be any directed edge.

(i) \((u, v)\) satisfies Condition **L1** if

\[
 f(V_{-v}(u) \cup \{v\}, v) \geq f(V_{-v}(u), u) > T. \tag{4}
\]

(ii) \((u, v)\) satisfies Condition **L2** if

\[
 f(V_{-v}) (u, u) \leq f(V_{-v}(u) \cup \{v\}, v) \leq T. \tag{5}
\]

**Definition 20** (*Figure 13*) Let \(u\) be a neighbor of \(v\) and \(u'\) a neighbor of \(v'\) Then

- Directed edge \((u', v')\) is above directed edge \((u, v)\) if \(v\) is on path \(\Pi(u, u')\) and \(u'\) is on path \(\Pi(v, v')\).
- Directed edge \((u', v')\) is below directed edge \((u, v)\) if \((u, v)\) is above \((u', v')\).

**Lemma 9** Let \((u, v)\) be a directed edge. Then, one of the following three cases must hold, with the corresponding consequences.

---

3 It thus needs to check each edge in the tree twice; once in each direction.
(i) \((u, v)\) satisfies L1. Then all edges \((u', v')\) above \((u, v)\) satisfy L1.

(ii) \((u, v)\) satisfies L2. Then all edges \((u', v')\) below \((u, v)\) satisfy L2.

(iii) \((u, v)\) does not satisfy L1 or L2. Then (a) \((u, v)\) satisfies the peaking criterion, (b) all edges \((u', v')\) above \((u, v)\) satisfy L1 and (c) all edges \((u', v')\) below \((u, v)\) satisfy L2.

Proof: Follows immediately from the definitions and path monotonicity. \(\square\)

Lemma 10: If at anytime during the first peaking phase edge \((u', v')\) satisfies either L1 or L2, \((u', v')\) will never satisfy the peaking condition anytime later during the first peaking phase.

Proof: First suppose that \((u', v')\) satisfied L1 at some time. If \(T_{u'}(u')\) never changes during the phase then (4) will remain satisfied and \((u', v')\) will never satisfy the peaking criterion. \(T_{u'}(u')\) can only change during the phase if a sink is committed inside \(V_{u'}(u)\). But sinks are never removed during the phase so if a sink is placed in \(V_{u'}(u)\), \((u', v')\) will still not be able to satisfy the peaking criterion during the phase.

Now suppose that \((u', v')\) satisfies L2 at some time. If \(T_{u'}(u')\) never changes during the phase then (5) will remain satisfied and \((u', v')\) will never satisfy the peaking criterion nor will any edge below it. The only way for \(T_{u'}(u')\) to change in this case is for some sink to be placed above it and remove \(T_{u'}(u')\). But, once it is removed \((u, v)\) will obviously never again satisfy the peaking condition. \(\square\)

At the start of the first peaking phase all edges will be initialized and marked as U(unknown).

Whenever a \((u, v)\) is tested for the peaking criterion, one of the three cases in Lemma 9 will occur. If case (i), label all edges \((u', v')\) above \((u, v)\) as L1. If case (ii), label all edges \((u', v')\) below \((u, v)\) as L2. If case (iii) do both before removing the edge and committing \(T_{u'}(u)\) to u.

More specifically, after labelling an edge L1, use Breadth-First Search to label all edges above it as L1 as well. If the procedure ever encounters an edge already labelled L1 it does not continue past that edge (since all of the edges above it were already labelled L1). Thus the total time to mark edges as L1 in the phase is O(n). A similar analysis shows that the total time required to mark edges as L2 in the phase is also O(n).

The algorithm will check the edges \((u, v)\) in a special order to be defined below. When checking an edge \((u, v)\) it first checks whether it is already marked as L1 or L2. If it is, it skips it since from Lemma 10, it doesn’t satisfy the peaking criterion. Only if \((u, v)\) is still marked U does the algorithm actually run the oracle to evaluate \(f(V_{u'}(u), u)\) and \(f(V_{u'}(u) \cup v, v)\). After completing the calculation it marks further edges using Lemma 9 and then performs a commit if required.

Recall that the centroid \(\rho(T)\) of a tree T with n nodes is a node u such that all subtrees falling off of u contain \(\leq n/2\) nodes. A centroid exists and can be found in \(O(n)\) time [25]. The algorithm will use a standard recursive centroid decomposition process to specify the edge checking order.

The process creates two sequences \(F_i\) and \(L_i\), containing, respectively, forests of trees, and sets of vertices. For node \(u \in V_{in}\), let \(\hat{N}(u)\) denote the set of neighbors of u in the full working tree \(T = T_{in}\).
Stage $i = 0$: Set $F_0 = \{T_{in}\}$.
Stage $i$, $i > 0$: Initialize $L_i := \emptyset$ and $F_i = \emptyset$.

For every tree $T'$ in the forest $F_{i-1}$,
  Remove $\rho(T')$ from $T'$, resulting in a forest of subtrees.
  Move the resulting forest of subtrees into $F_i$.
  Add $\rho(T')$ into $L_i$.

This process terminates when $F_i$ is empty. Note that every $v \in V$ is chosen as the centroid of exactly one tree in this process so the $L_i$ are a partition of $V$.

Set $W_j = \bigcup_{i=1}^{j} L_i$. Note that $W_t = V_{in}$.

From the definition of a centroid, trees in $F_i$ all have size $\leq n/2^i$ so $t = O(\log n)$.

The first peaking phase processes the edges in $E$ by examining $i = 1, 2, \ldots, t$ in order and, for every $v \in L_i$, checking all $(u, v)$ where $u \in N(v)$. Since $W_i = V$, this checks all edges.

See Algorithm 2. When edge $(u, v)$ is encountered in line 5, the algorithm first determines if it is still marked $U$. If it is, the algorithm saves it in set $E'_i$ and performs the appropriate oracle calls but defers the actual checking of the peaking condition to later in the stage.\footnote{$(u, v)$ could have been checked immediately. The deferment is introduced to simplify the later use of parametric searching in Sect. 5.}

Lines 12-14 check for the degenerate case in which the current tree centroid $v = \rho(T')$ can support $T_{in}$. Since $T$ can be served by just the one sink $v$, the algorithm terminates.

Otherwise, the algorithm examines the results of the oracle calls on edges in $E'_i$, applying Lemma 9 to appropriately label edges as $L1$ or $L2$ and then applying the Peaking Lemma to the edges that satisfy the criterion to create new sinks and commit blocks to them.

We now examine the running time of Algorithm 2. We already saw that the total cost of labelling edges is $O(n)$. The remainder of the algorithm with the exception of line 8 can also be implemented in $O(n)$ time. Now define

$$C_i(u, v) = \begin{cases} V_{-v}(u) & \text{if } (u, v) \text{ was added to } E'_i \text{ in line 7.} \\ \emptyset & \text{otherwise.} \end{cases}$$

This is well defined since every edge appears in each $E'_i$ at most once. By definition, the cost of implementing line 8 for $(u, v)$ is $O(t_A(|C(u, v)| + 1))$.

We first prove a utility lemma.

**Lemma 11** Let $T_1$ and $T_2$ be two trees in $F_{i-1}$, $v_1 = \rho(T_1)$, $v_2 = \rho(T_2)$ be their centroids and $u_1 \in N(v_1)$, $u_2 \in N(v_2)$. Then

$$C_i(u_1, v_1) \cap C_i(u_2, v_2) = \emptyset. \quad (6)$$

In the statement of this lemma, it is possible that $T_1 = T_2$.\footnote{$(u, v)$ could have been checked immediately. The deferment is introduced to simplify the later use of parametric searching in Sect. 5.}
**Algorithm 2** Processing the edges, stage by stage

1: Perform the tree centroid decomposition process defined in the text
2: \[\text{for } i = 1 \text{ to } t \text{ do} \quad \triangleright \text{Stage } i. \text{Checks } (u, v) \text{ for } v \in L_i.\]
3: \[\text{Set } E'_i = \emptyset\]
4: \[\text{for all } v \in L_i \text{ do} \quad \triangleright \text{Process all centroids } v = \rho(T') \text{ for some } T' \in \mathcal{F}_{i-1}\]
5: \[\text{for all } u \in N(v) \text{ do}\]
6: \[\text{if } (u, v) \text{ is marked } U \text{ then}\]
7: \[\text{Add } (u, v) \text{ to } E'_i\]
8: \[\text{Evaluate } a(u, v) = f(V_u(u), u), \text{ and } b(u, v) = f(V_v(u) \cup \{v\}, v)\]
9: \[\text{end if}\]
10: \[\text{end for}\]
11: \[\text{end for}\]
12: \[\text{for all } v \in L_i \text{ do} \quad \triangleright \text{Checks for degeneracy condition (7)}\]
13: \[\text{if } \forall u \in N(v), (u, v) \in E'_i \text{ and } b(u, v) \leq T \text{ then}\]
14: \[\text{Commit } V \text{ to } v \text{ and terminate the algorithm.}\]
15: \[\text{end if}\]
16: \[\text{end for}\]
17: \[\text{for } (u, v) \in E'_i \text{ do} \quad \triangleright \text{Mark edges appropriately and check peaking condition}\]
18: \[\text{Apply Lemma 9 appropriately based on whether } a(u, v) \leq T \text{ and } b(u, v) > T\]
19: \[\text{if } a(u, v) \leq T \text{ and } b(u, v) > T \text{ then} \quad \triangleright (u, v) \text{ satisfies peaking Criterion}\]
20: \[\text{apply Peaking Lemma to commit } V_v(u) \text{ to } u\]
21: \[\text{end if}\]
22: \[\text{end for}\]
23: \[\text{end for}\]

**Proof** If \(i = 1\) the lemma is trivially true since \(\mathcal{F}_0 = \{T\}\) so \(v_1 = v_2\) and thus \(V_{v_1}(u_1)\) and \(V_{v_1}(u_2)\) are disjoint.

We therefore assume that \(i > 1\). We also assume that \(v_1 \neq v_2\) since otherwise \(V_{v_1}(u_1)\) and \(V_{v_1}(u_2)\) are obviously disjoint.

We finally assume that

\[
\text{There does not exist } w \in W_{i-1} \text{ s.t. } \forall z \in N(w), \ f(V_{w}(z) \cup \{w\}, w) \leq T. \quad (7)
\]

This is because if such a \(w\) existed then lines 12-14 in Algorithm 2 would have terminated the algorithm before the start of stage \(i\).

Observe that all nodes in \(V\) lie in \(T_{in}\) during stage 0 but by the end of stage \(i - 1\), \(T_1\) and \(T_2\) are disconnected. By construction there must exist some node \(w \in W_{i-1}\) (whose removal disconnected \(T_1\) and \(T_2\)) that lies on the path \(\Pi(v_1, v_2)\) connecting \(v_1\) and \(v_2\).

- (7) implies \(\exists z \in N(w)\) satisfying \(f(V_{w}(z) \cup \{w\}, w) > T\).
- If \(v_1 \notin T_{v_2}(u_2)\) and \(v_2 \notin T_{v_1}(u_1)\) then \(V_{v_1}(u_1) \cap V_{v_1}(u_1) = \emptyset\) so (6) is trivially true. (Fig. 14)
- If either of \((u_1, v_1)\) or \((u_2, v_2)\) were labelled \(L_1\) or \(L_2\) at the end of stage \(i - 1\) then (6) would be trivially true.

If the Lemma is incorrect we may therefore assume that neither \((u_1, v_1)\) or \((u_2, v_2)\) were labelled \(L_1\) or \(L_2\) at the end of stage \(i - 1\) and at least one of \(v_1 \in T_{v_2}(u_2)\) or \(v_2 \in T_{v_1}(u_1)\) is true. WLOG assume that \(v_2 \in T_{v_1}(u_1)\).

Figure 15 Label the neighbors of \(w\) so that \(z_1\) is on the path from \(w\) to \(v_1\), \(z_2\) is on the path from \(w\) to \(v_2\) and \(z_3, \ldots, z_s\) are the others (if they exist).
Note that, if, for any \(j > 1\), \(f(V - w(z_j) \cup \{w\}, w) > T\) then since \((u_1, v_1)\) is above \((z_j, w)\), \((u_1, v_1)\) would have been labelled \(\mathbf{L1}\) by the end of stage \(i - 1\) which we assumed was not the case. Thus, for all \(j > 1\), \(f(V - w(z_j) \cup \{w\}, w) \leq T\). (7) then implies \(f(V - w(z_1) \cup \{w\}, w) > T\), contradicting the result of the previous paragraph.

Next note that if \(v_1 \in T_{-v_2}(u_2)\) the exact same argument would show that for all \(j \neq 2\), \(f(V - w(z_j) \cup \{w\}, w) \leq T\), otherwise \((u_1, v_1)\) would have been labelled \(\mathbf{L1}\) by the end of stage \(i - 1\), which we assumed was not the case. In particular this would imply \(f(V - w(z_1) \cup \{w\}, w) \leq T\), contradicting the result of the previous paragraph.

Thus \(v_2 \in T_{-v_1}(u_1)\) and \(v_1 \notin T_{-v_2}(u_2)\). But this and the fact that \(f(V - w(z_2) \cup \{w\}, w) \leq T\) immediately imply that \((u_2, v_2)\), which is below \((z_2, w)\), would have been labelled \(\mathbf{L2}\) by the end of stage \(i - 1\), contradicting our assumptions. \(\square\)

We now prove

**Lemma 12**

1. Algorithm 2 works in \(O(\log n)\) stages with the specific oracle calls made during stage \(i\) only dependent upon the results of the oracle calls made in stages \(j < i\) and not on the results of any oracle calls during stage \(i\).
2. In each stage the total work performed by the oracle calls is \(O(t_A(n))\).
3. The total amount of work performed by Algorithm 2 is $O\left( t_A(n) \log n \right)$.

Proof (1) is from the definition of the algorithm.

For (2) Let $E_i$ be the edges processed in stage $i$, i.e., $(u, v) \in E_i$ if $v \in L_i$. By definition, $E'_i \subseteq E_i$. From Lemma 11 no vertex $w \in V$ can appear in more than one set $C(u, v)$ for $(u, v) \in E_i$. Thus

$$
\sum_{(u,v) \in E'_i} |C(u, v)| \leq \sum_{(u,v) \in E_i} |C(u, v)| \leq n.
$$

So, by asymptotic subadditivity, the total amount of work done in stage $i$ in line 8 will be

$$
O\left( \sum_{(u,v) \in E_i} t_A(|C(u, v)| + 1) \right) = O\left( \sum_{(u,v) \in E_i} O\left( t_A(|C(u, v)|) \right) \right) = O\left( t_A(n) \right)
$$

proving (2).

(3) then follows from the fact that $t = O(\log n)$, the remainder of the work in Algorithm 2 outside of line 8 is $O(n)$ and only another $O(n \log n)$ time is required for the decomposition. \hfill \square

4.1.2 Later Peaking Phases by Binary Search

All later peaking phases start immediately after a reaching phase has completed by finding an edge $(u, v)$ satisfying the reaching criterion. Lemmas 7 and 8 state that, after the reaching phase, the remaining tree contains at most one edge satisfying the peaking condition. They also provide necessary and sufficient conditions for that edge to exist and state that that edge must lie along a particular path, either $\Pi(v, h)$ or $\Pi(v, h')$. In both cases this permits binary searching along the path to find the peaking edge. This is formalized in Algorithm 3. The algorithm finds this edge, if it exists. After the algorithm complete,s no edge satisfies the peaking criterion, so the remaining tree is RC-viable and a new reaching phase starts.

The procedure performs $O(n)$ book-keeping, $O(1)$ oracle calls and, possibly, one binary search requiring an additional $O(\log n)$ oracle calls. Thus

Lemma 13 Each individual peaking phase after the first one can be implemented using only $O(\log n)$ oracle calls and $O(t_A(n) \log n)$ time.

4.2 Creating and Maintaining the Hub Tree

At the start of each reaching phase the algorithm must construct the current hub tree. This entails identifying an appropriate root $r$, the hub nodes $V_H(ST)$ and for each hub node $v$, pointers from $v$ to its children and to $p(v)$ and $p_H(v)$. For each $v$ it will also need to find the set $S(v)$ of sinks in $T(v)$, the directed subtree of $T$ rooted at $v$, that can support the bulk path from $s$ to $v$. 

Springer
Algorithm 3 Peaking Phase after Reaching Phase

1: \( \triangleright (u, v) \) satisfies the reaching condition in \( T_H(S_T) \) and \( V_{\rightarrow u}(u) \) has been removed
2: \( \triangleright \) Checks the scenarios from Lemmas 7 and 8.
3: \( \triangleright \) Let \( h, v', v'', h' \) be as introduced in Lemmas 7 and 8.
4: \( \triangleright \) After Completion, resulting tree is RC-viable.
5: return.
6: end procedure
7: \( \triangleright h = r \) and \( r \) had exactly two children in \( T_H(S_T) \). Lemma 8
8: end if
9: end if
10: end if
11: end procedure
12: return.
13: end if
14: if \( (h \neq r) \) OR \( (h = r \) and \( r \) had at least three children in \( T_H(S_T) ) \) then \( \triangleright \) Lemma 7
15: if \( h \neq v(v') \) AND \( (f(V_{\rightarrow h}(v')) \cup \{h\}, h) > T \) then
16: \((ü, \bar{v}) = PSearch(v, h)
17: Commit \( V_{\rightarrow ü}(ü) \) to \( ü\)
18: end if
19: else \( \triangleright h = r \) and \( r \) had exactly two children in \( T_H(S_T) \). Lemma 8
20: if \( (f(V_{\rightarrow h}(v'')) \cup \{h'\}, h') > T \) then
21: \((ü, \bar{v}) = PSearch(v, h'')
22: Commit \( V_{\rightarrow ü}(ü) \) to \( ü\)
23: end if
24: end if

Algorithm 4 Parallel Binary Search in Reaching Phase

1: \( \triangleright \) The structure \( T_H(S_T) \) with root \( r \) is given. \( S_T \) is associated sink set.
2: Create the paths \( \Pi(s, r) \) in \( T_H(S_T) \), written as \( s = v_1 \), \( v_2 \), \ldots, \( v_s = r \).
3: \( \triangleright \forall \bar{s} \in S_T \), Binary search to find largest \( i \) \( \in [1, t_s - 1] \) such that \( f(BP(v_i^s), s) \leq T \)
4: \( \triangleright \) At termination, \( m_s \) is largest index; \( u_s = v_{m_s}^s \) is highest such node on \( \Pi(s, r) \)
5: \( \forall \bar{s} \in S_T \), set \( \ell_s = 1, q_s = t_s, m_s = \lceil (\ell_s + q_s)/2 \rceil \)
6: while \( \exists \bar{s} \in S_T \) such that \( \ell_s \neq q_s \) do \( \triangleright \) Perform the binary searches in Parallel
7: for \( s \in S_T \) such that \( \ell_s \neq q_s \)
8: Evaluate \( a_s = f(BP(v_{m_s}^s), s) \)
9: end for
10: for \( s \in S_T \) such that \( \ell_s \neq q_s \)
11: if \( a_s \leq T \) then
12: \( \ell_s = m_s \)
13: else
14: \( q_s = m_s \)
15: end if
16: end for
17: end while

Definition 21 Set

\[ S(v) = \{ s \in S \cap V(v) : f(BP(v, s), s) \leq T \} \]

Everything except for the calculation of the \( S(v) \) can be easily done in \( O(n) \) time, and does not require any oracle calls.

To calculate \( S(v) \), assume the hub-tree structure has been built and let \( s \in S_T \). \( f(BP(u, s), s) \) is a non-decreasing function as \( u \) moves up the tree path \( \Pi(s, r) \).
Thus, a binary search using $O(\log n)$ oracle $A$ calls can find the highest node $u_s$ on $\Pi(s, r)$ satisfying $f(BP(u_s, s), s) \leq T$. This binary search uses $O(t_A(n) \log n)$ time.

This is shown in Algorithm 4, which does the binary searches for all of the $s \in S_T$ in parallel. Since $|S_T| \leq k$, the entire procedure takes $k \log n t_A(n)$ time. Running the binary searches in parallel will help reduce the running time later in the parametric search version used in the optimality procedure.

After finding $u_s$ the algorithm walks up the path $\Pi(s, u_s)$ adding $s$ to $S(v)$, for every node $v$ on the path. This requires an extra $O(nk)$ time in total. For each $v \in V_H(S_T)$, we maintain the list $S(v)$ of sinks partitioned into sublists; each sublist is associated with the hub child of $v$ that contains those sinks. Combining all of the above, the time required for constructing a hub tree is $O(nk + k \log n t_A(n)) = O(k \log n t_A(n))$.

### 4.3 Implementing the Reaching Phase

Assume that the hub tree is given along with its associated $S(v)$ lists. The self-sufficiency tests in Sect. 3.4.1 can now be restated in terms of $S(v)$.

**Lemma 14** Let $v \in V_H(S_T)$ be a non-hub node, $u$ its unique child in $T_H(S_T)$ and $T(v)$ the subtree of $T$ rooted at $v$. If $T_{-v}(u)$ is recursively self-sufficient then one of the following two cases must occur:

(i) $|S(v)| = 0$ and $(u, v)$ satisfies the reaching criterion.

(ii) $|S'(v)| > 0$ and $T(v)$ is recursively self-sufficient with every sink in $S(v)$ as a witness to its self-sufficiency.

**Proof** This lemma is essentially a restatement of Corollary 2 rewritten for the special case in which $v$ only has the one child $u$ in $T_H(S_T)$.

**Lemma 15** Let $v$ be a non-sink hub in $T_H(S_T)$, $u_1, \ldots, u_t$ be its hub-children and all the $T_{-v}(u_i)$ are recursively self-sufficient. Then one of the following two cases must occur:

(i) $\exists i$, such that $S(u_i) \cap S(v) = \emptyset$

$\Rightarrow (u_i, v)$ satisfies the reaching criterion.

(ii) $\forall i, S(u_i) \cap S(v) \neq \emptyset$

$\Rightarrow T(v)$ is recursively self-sufficient with every sink in $S(v)$ as a witness to its self-sufficiency.

**Proof** This lemma is essentially a restatement of Corollary 2 rewritten for the case when $v$ has more than one child in $T_H(S_T)$.

Now combine the pieces.

First, in $O(|V_H(S_T)|) = O(n)$ time, preprocess the nodes in $V_H(S_T) \setminus S_T$ by topologically sorting them so that if $v_i$ is the child of $v_j$ then $i < j$.

Next, process the nodes in $V_H(S_T)$ in this topological order. This will ensure that a node will be processed only after its hub-children have already been processed. By induction, after a node $v$ has been processed, if the algorithm hasn’t halted, $T(v)$ will be recursively self sufficient.
Algorithm 5 Reaching Stage

1: \( V_{H(S_T)} \setminus S_T \) is topologically sorted; if \( v_i \) is the child of \( v_j \) then \( i < j \).
2: \( t = |V_{H(S_T)} \setminus S_T| \).
3: If line 24 is reached without Break Out, then \( T \) is self-sufficient
4: 
5: for \( i = 1 \) to \( t \) do
6: \( v := v_i \)
7: if \( v \) is a non-sink hub then \( \triangleright \) Apply Lemma 15
8: for \( u \) a child of \( v \) in \( T_H(S_T) \) do
9: if \( S(u) \cap S(v) = \emptyset \) then \( \triangleright (u, v) \) satisfies Reaching Criterion
10: Remove \( V(u) \) from \( T \)
11: Commit blocks for \( V(u) \) to sinks in \( V(u) \cap S \) using Lemma 5
12: Break out of Procedure
13: end if
14: end for
15: else \( \triangleright v \) is not a hub. Apply Lemma 14
16: Set \( u \) to be the unique hub-child of \( v \)
17: if \( |S(u)| > 0 \) and \( |S(v)| = 0 \) then \( \triangleright (u, v) \) satisfies Reaching Criterion
18: Remove \( V(u) \) from \( T \)
19: Commit blocks for \( V(u) \) to sinks in \( V(u) \cap S \) using Lemma 5
20: Break out of Procedure
21: end if
22: end if \( \triangleright \) If no Break out, \( S(v) \) are witnesses to recursive self-sufficiency of \( T(v) \)
23: end for
24: Commit all of \( T \) to \( S \) and terminate algorithm. \( \triangleright \) Entire \( T \) is recursively self-sufficient

Processing a non-sink hub node \( v \) uses Lemma 15 to check if any of the edges leading to \( u_i \) satisfy the reaching criterion. If yes, the algorithm commits the proper nodes to sinks in \( O(n) \) time and exits. Otherwise the tree rooted at \( v \) will be recursively-self sufficient and the algorithm continues.

Processing a non-hub node \( v \) uses Lemma 14 to check in \( O(1) \) time if \((u, v)\) satisfies the reaching criterion, where \( u \) is \( v \)’s unique hub child. If yes, the algorithm commits the proper nodes to sinks in \( O(n) \) time and exits. Otherwise \( v \) will be recursively self sufficient and the algorithm continues.

If the algorithm completes the entire For loop and reaches line 24, then the entire tree \( T \) is recursively self-sufficient so \( T \) can be fully committed to \( S \) and the algorithm terminates.

Lines 9 and 17 can be implemented in \( O(1) \) time because of the way the lists were stored.

We have therefore just proven

**Lemma 16** If the hub tree with its associated \( S(v) \) lists is already given, then the reaching phase can be implemented in \( O(n) \) time.

### 4.4 Combining the Pieces

This section has shown how to implement the entire bounded cost algorithm. It follows the generic structure of Algorithm 1, alternating Peaking and Reaching Phases.

The actual work was done by four logically distinct parts listed below. This decomposition will permit the parametric search extension in the next section.
1. **The First Peaking Phase**
   - Implemented using tree centroid decomposition method of Sect. 4.1.1.
   - Divided into $O(\log n)$ stages. Each stage performs $O(n)$ extra work plus one amortized Oracle call, requiring $O(t_A(n)))$ time.
   - Total time required $O(\log n t_A(n)))$.
   - Number of actual oracle calls made could be as high as $\Theta(n)$.

2. **All other Peaking Phases**
   - Implemented using binary search method of Sect. 4.1.2
   - Uses $O(\log n t_A(n))$ time with $O(\log n)$ oracle calls per peaking phase.
   - At most $k-1$ peaking phases; $O(k \log n t_A(n))$ total time for all peaking phases.

3. **Creating Hub Tree after Peaking phase**
   - Implemented using parallel binary search method of Sect. 4.2
   - $O(\log n)$ parallel steps, each making $O(k)$ oracle calls.
   - Uses $O(k \log n)$ total oracle calls. Total time required $O(k \log n t_A(n))$
   - At most $k$ peaking stages; Using total $O(k^2 \log n)$ total oracle call and $O(k^2 \log n t_A(n))$ total time.

4. **Reaching Phases**
   - Implemented using Algorithm 5
   - Uses $O(n)$ time with no oracle calls per each reaching phase
   - Assumes pre-existing hub tree with preconstructed lists lists $S(u)$
   - At most $k$ reaching stages; $O(nk)$ total time for all reaching stages

Combining these parts proves Theorem 2. For later use we denote this complete algorithm for solving the bounded-cost minmax $k$-sink problem as $B$ and its running time on an input of size $n$ as $B(n) = O(k^2 t_A(n) \log n)$.

5 **Full Problem: Cost Minimization via Parametric Searching**

By binary searching over all possible values of $T$ and using $B$ to test the feasibility of these $T$, it is straightforward to construct a weakly polynomial time algorithm for the general minmax $k$-sink problem of finding $T^*$, the smallest $T$ for which $k$ sinks suffice.

Modifying $B$ to produce a strongly polynomial time algorithm, as in Theorem 3, will require using a variation on Megiddo’s parametric searching technique [31].

**Definition 22** The State of algorithm $B$ at any given time will be the current $(S_{\text{out}}, P_{\text{out}})$, working tree $T$, the edge labels in the first peaking phase and the $S(v)$ values in the hub trees.

Note that all of the information saved by $B$, i.e., $S$, $S_T$ and the rest of the hub tree information, can be directly constructed from its state. Thus if two invocations of $B$
Single-Query:
1. Perform the evaluation \( a := f(U,s) \) in \( O(t_A(n)) \) time
2. Reset \( T^L, T^H \) if necessary as follows:
   - If \( a \leq T^L \) or \( a \geq T^H \), do nothing
   - If \( a \in (T^L, T^H) \), run a separate version of \( B \) from scratch with value \( T = a \), and
     - If \( a \) is infeasible (\( B \) answers ‘No’), set \( T^L = a \).
     - If \( a \) is feasible (\( B \) answers ‘Yes’), set \( T^H = a \).
3. Resolve the query \( a \leq T \) in \( I \) by returning the answer to \( a \leq T^L \)

**Fig. 16** A single-query in Algorithm \( I \). Requires \( O(B(n)) \) time

on two different values \( T' \) and \( T'' \) both stop mid-calculation in the same state there is no way to distinguish between them.

In the parametric search version, \( T \) will no longer be a constant; instead we interfere with the normal course of \( B \) by changing \( T \) during runtime.

This interfered version is denoted as Algorithm \( I \). The decision to interfere is based on a threshold range \((T^L, T^H)\). \( I \) starts with \((T^L, T^H) = (0, +\infty)\) and always maintains the following invariants:

\[(I1) \quad T^L < T^H.\]
\[(I2) \quad T^L \text{ never decreases and } T^H \text{ never increases.}\]
\[(I3) \quad T^L \text{ will be infeasible and } T^H \text{ will be feasible.}\]
\[(I4) \quad \text{At each step of } I, \text{ the corresponding state of } B \text{ would be identical for all values of } T \in [T^L, T^H]. \text{(Note the flipping of open and closed intervals.)}\]

Intuitively, \( I \) “pretends” that it is running \( B \) for all \( T \in [T^L, T^H) \) while pruning away “useless values”. We will soon see that the properties above will imply that \( I \) terminates with \( T^* = T^H \) being the correct solution.

Oracle calls may occur in two different formats during the running of \( B \).

**Single-Queries:** See Fig. 16.

The first format for oracle calls is to set \( \alpha = f(U,s) \) for some specific tree \( U \) and vertex \( s \in U \), and then evaluate whether \( \alpha \leq T \) or \( \alpha > T \). This occurs in Algorithm 3 (Peaking Phase after Reaching Phase), both in the Psearch procedure and in lines 15 and 20.

The corresponding version in \( I \) will be called a single-query. It appropriately updates \( T^L \) or \( T^H \), if necessary, and then responds as if the answer to \( \alpha \leq T \) is the answer to \( \alpha \leq T^L \). The time required for a single query is then

\[
S(n) = O(t_A(n) + B(n)) = O(B(n));
\]

**Group-Queries:** See Fig. 17.

The second format for oracle calls is, for \( i = 1, \ldots, m \), to first evaluate \( \alpha_i = f(U_i, s_i) \) for trees \( U_i \) and vertices \( s_i \in U_i \), and then, after those evaluations, deciding, for each \( i \), whether \( \alpha_i \leq T \) or \( \alpha_i > T \). The corresponding version in \( I \) algorithm will be called a group-query.

\[\text{The version of parametric searching used here is specialized for the case in which the feasibility test } B \text{ can only determine whether } T^* \leq T \text{ but not whether } T^* = T. \text{ See [1, p. 415] for more details.}\]
Group-Query:
1. Perform the evaluations \( a_i := f(U_i, s_i) \), \( i = 1, \ldots, m \) and sort them in \( O(m \log m) \) time.
2. Relabel the \( a_i \) so that \( a_1 \leq a_2 \leq \cdots \leq a_m \). Set \( V = \{ a_1, a_2, \ldots, a_m \} \).

   Note that feasibility of any \( a_i \) can be tested in \( O(n) \) time by running a separate version of \( B \) with \( T = a_i \).
3. - If \( a_1 \) is feasible, i.e., all of the \( a_i \) are feasible, set \( a^L = T^L \).
   - If \( a_m \) is not feasible, i.e., all of the \( a_i \) are not feasible, set \( a^H = T^H \).
   - Otherwise, in \( O(B(n) \log m) = O(B(n) \log m) \) time, binary search in \( V \) to find
     \[ a^L = \max\{ a_i \in V : a_i \text{ is not feasible} \} \quad a^H = \min\{ a_i \in V : a_i \text{ is feasible} \} \]
4. Set \( T^L = \max\{ a^L, T^L \} \), \( T^H = \min\{ a^H, T^H \} \)
5. For all \( i \), resolve the queries \( a_i \leq T \) in \( \mathcal{I} \) by returning the answer to \( a_i \leq T^L \)

Fig. 17 A group-query in Algorithm \( \mathcal{I} \). Requires \( O(B(n) \log m) \) time plus the time for line 1

This first calculates all the \( a_i = f(U_i, s_i) \). It then sorts the \( a_i \) by value and binary searches to find the largest infeasible and smallest feasible \( a_i \). Each query in the binary search is a feasibility test \( B \) so this requires \( O(B(n) \log m) \) time. It then appropriately updates \( T^L \) or \( T^H \), if necessary, and responds to each query \( a_i \leq T \) by returning the answer to \( a_i \leq T^L \).

This type of group-query occurs in two places in \( B \). The first place is as a stage, in lines 3-22, in Algorithm 2. In this case the group consists of the values \( a^L(u, v), b(u, v) \), and has size \( m = O(n) \). Recall that the cost of evaluating \( a^L(u, v), b(u, v) \) was shown via an amortization argument to be \( O(t_A(n)) \). So, the total group-query cost of a stage is

\[ GS(n) = O(B(n) \log n + t_A(n)) = O(B(n) \log n). \quad (9) \]

The second place that group-queries occur is in the reaching phase when finding the highest nodes \( u_s \) on \( \Pi(s, r) \) satisfying \( f(BP(u_s, s), s) \leq T \). This is lines 8-18 in Algorithm 4. In this case the group consists of the values \( a_s \) for \( s \in S_T \) such that \( \ell_s \neq q_s \). This group has size \( O(|ST|) = O(k) \). So, the total cost of one group-query during a reaching phase will be

\[ GR(n) = O(B(n) \log k + kt_A(n)) = O(B(n) \log k). \quad (10) \]

5.1 The Actual (Interfered) Algorithm \( \mathcal{I} \).

We now describe \( \mathcal{I} \), the algorithm for solving the Minmax \( k \)-sink problem. This algorithm starts with \( [T^L, T^H] = (0, +\infty] \) and runs \( B \). Every time \( B \) encounters a single-query it runs the code in Fig. 16, replacing the query \( a \leq T \) with \( a \leq T^L \).

Every time \( B \) encounters a group-query it runs the code in Fig. 17 replacing the queries \( a_i \leq T \) with \( a_i \leq T^L \). A step in the algorithm will start immediately after running a single or group query and stop immediately before running the next one.

We claim that, after \( \mathcal{I} \) terminates, \( T^* = T^H \) is the optimal value. To get the final tree-decomposition we then rerun \( B \) with \( T = T^H \). More explicitly
Lemma 17 Let $T^*$ be the optimal value of $T$ and let $(T^{L_i}, T^{H_i})$ be the threshold range immediately before running the $i$th step (either a single or group step) of algorithm $\mathcal{I}$. Let $(T^<, T^>)$ be the threshold range when $\mathcal{I}$ terminates. Then

1. $\forall i, T^* \in (T^{L_i}, T^{H_i})$.
2. Let $T' \in [T^{L_i}, T^{H_i})$. Then, the state of Algorithm $B$ with $T = T'$ immediately before running its $i$th step will be in exactly the same as the state of algorithm $B$ with $T = T'$ immediately before running its $i$th step.
3. Algorithm $\mathcal{I}$ will end in exactly the same state as as algorithm $B$ run with $T = T^<$.

Proof (1) $T^* \leq T^{H_i}$ because $T^{H_i}$ is always set to be a feasible value of $T$. Similarly, $T^* > T^{L_i}$ because $T^{L_i}$ is always set to be a non-feasible value of $T$.

(2) Will be proven by induction on $i$.

The process starts with $(T^{L_1}, T^{H_1}) = (0, \infty)$. As long as no queries have been made the algorithm’s state will be the same regardless of the value of $T$. Thus (2) is valid for $i = 1$.

Now assume (2) is correct for $i$. From the induction hypothesis, the state of $B$ for $T = T^{L_i}$ and $T = T'$ will be identical immediately before the $i$th query occurs.

First, assume that the next query is a single query against some value $a$.

Note that, after completing Line 2 in the Single-query, $a \notin (T^{L_{i+1}}, T^{H_{i+1}})$. So, if $T' \in (T^{L_{i+1}}, T^{H_{i+1}})$,

$$a \leq T^{L_{i+1}} \Leftrightarrow a \leq T'$$

Thus Algorithm $B$ can not distinguish between the answer to $a_j \leq T^{L_{i+1}}$ and $a_j \leq T'$. Since the decisions made by $B$ before the next query only depend upon the prior labels of edges and the results of the $a \leq T$ query, $B$ will continue to behave identically for both $T = T^{L_{i+1}}$ and $T = T'$ until immediately before the $(i+1)$st query, proving (2) is correct for $i + 1$ in the single-query case.

Next, assume that the next query is a group-query against some values $a_j , j = 1, \ldots, m$. Note that, after completing Line 4 in the group-query, $\forall j, a_j \notin (T^{L_{i+1}}, T^{H_{i+1}})$. So, if $T' \in (T^{L_{i+1}}, T^{H_{i+1}})$,

$$a_j \leq T^{L_{i+1}} \Leftrightarrow a_j \leq T'$$

Thus Algorithm $B$ can not distinguish between the answer to $a_j \leq T^{L_{i+1}}$ and $a_j \leq T'$. Since the decisions made by $B$ before the next query only depend upon the prior labels of edges and the results of the $a_j \leq T$ queries, $B$ will continue to behave identically for both $T = T^{L_{i+1}}$ and $T = T'$ until immediately before the $(i+1)$st query, proving (2) is correct for $i + 1$ in the single-query case.

(3) follows from the analysis of (2) and the fact that $\forall i, T^L \in [T^{L_i}, T^{H_i})$.

Lemma 18 1. Let $(T^<, T^>)$ be the threshold range when $\mathcal{I}$ terminates and $T^*$ be the optimal value of $T$. Then $T^* = T^>$.

2. In particular, running the bounded cost Algorithm $B$ on $T := T^>$ will retrieve the optimal feasible configuration.

3. Algorithm $\mathcal{I}$ will terminate in $T(n) = O(k \log k, \log n)k^2nA(n)) \log^2$ time.
Proof From Lemma 17 (3), the number and type of steps run by algorithm $I$ is exactly the same as those run by algorithm $B$ on $T^<$. Let $T' \in [T^<, T^>]$. From Lemma 17 (2), the number and type of steps run by algorithm $B$ on $T'$ is the same as the number and type of steps run by algorithm $B$ on $T^<$. In particular, since $T^<$ is infeasible, this means that $T'$ is infeasible. Thus $T^>$ is the smallest feasible value of $T$, i.e., $T^* = T^>$, proving 1. 2 follows automatically.

Lemma 17 (3) implies that the number and type of steps run by algorithm $I$ is exactly the same as those run by algorithm $B$ on $T^<$. The running time of $I$ is the time for running $B$ on $T^<$ plus the work done in the group-queries in the first peaking phase, the work done in the group queries in all of the reaching phases and the work done in all of the single queries in all except the first peaking phase.

There are $O(\log n)$ stages in the first peaking phase so the total work done in those group queries is $O(\log n GS(n))$.

Each reaching phase does $O(\log n)$ group queries and there are at most $k$ such phases so the total work done in those group queries is $O(k \log n GR(n))$.

Finally, each peaking phase except for the first does $O(\log n)$ single-queries and the total number of such phases is $O(k)$. Thus the total work done in all of those signal queries is $O(k \log n S(n))$.

Combining all of the above yields

$$T(n) = O\left( B(n) + k \log n S(n) + \log n GS(n) + k \log n GR(n) \right)$$
$$= O\left( B(n) + k \log n B(n) + \log n (B(n) \log n) + k \log n (B(n) \log k) \right)$$
$$= O\left( B(n) \left[ 1 + k \log n + \log^2 n + k \log k \log n \right] \right)$$
$$= O\left( B(n) \log n (\log n + k \log k) \right)$$
$$= O\left( \max(k \log k, \log n) k^2 t_A(n) \log^2 n \right).$$

Theorem 3 follows immediately from the previous Lemma.

Note: Straightforward application of parametric search to $B$ would require a call to $B$ every time the oracle $A$ was called. This first peaking phase can require as many as $\Theta(n)$ oracle calls, resulting in an $\Theta(nB(n)) = \Theta(n t_A(n))$ running time for that phase in the parametric search version. The centroid decomposition and distinction between Single and Group-Queries were used to replace this extra factor of $\Theta(n)$ by $\max(k \log k, \log n)$. 

\qed
Let \((u, v)\) be oriented so that it starts at \(u\) and ends at \(v\).
Then \(x < x'\) if and only if \(x \leq x'\) then
\[
f(V_{-v}(u) \cup \{x\}, x) \leq f(V_{-v}(u) \cup \{x', x\}')
\]

6 The Continuous Case

Until this point the analysis has always assumed the discrete version of the problem, in which sinks are required to be nodes in \(V_{in}\). This section will extend those results to the continuous case, in which sinks can be located on edges.

This first requires extending the definition of minmax monotone cost functions to edges.

**Definition 23** (Figure 18) Let \(T = (V, E)\) be a tree and \(f(\cdot, \cdot)\) a monotone minmax cost function as defined in Sect. 2.2.1.

For \(e = (u, v) \in E\), orient \(e\) so that it starts at \(u\) and ends at \(v\). Let \(x, x' \in e\). Denote

\[
\begin{align*}
x \leq x' & \quad \text{if and only if} \quad x \text{ is on the path from } u \text{ to } x', \\
x < x' & \quad \text{if and only if} \quad x \leq x' \text{ and } x \neq x'.
\end{align*}
\]

(1) \(f\) is continuous on \((u, v)\) if \(f(V_{-v}(u) \cup \{x\}, x)\) is a continuous function for \(u \leq x \leq v\).

(2) \(f\) is non-decreasing on \((u, v)\) if

\[
\forall x, x', u \leq x < x' \leq v, \quad f(V_{-v}(u) \cup \{x\}, x) \leq f(V_{-v}(u) \cup \{x', x\}')
\]

In the continuous case, minmax monotone cost functions must additionally be continuous and non-decreasing. This is the natural generalization of path-monotonicity.

Note: This definition is satisfied in the sink evacuation problem. Let \(d(x, v)\) denote the time required to travel from \(x\) to \(v\). It is natural to assume that this is a non-increasing continuous function in \(x\). Since flow travels smoothly without congestion inside an edge, if the last flow arrived at node \(v\) at time \(t\), then it had been at \(x > u\) at time \(t - d(x, v)\). Thus

\[
f(V_{-v}(u) \cup \{x\}, x) = f(V_{-v}(u) \cup \{v, v\}) - d(x, v) \quad (11)
\]

so condition (1) is satisfied and condition (2) is satisfied for every \(x\) except possibly \(x = u\). Now consider the time \(t'\) that the last flow arrives at node \(u\) and let \(t' + w\) be
the time that this last flow enters edge \((u, v)\). Since flow doesn’t encounter congestion inside an edge, it arrives at \(v\) at time \(t' + w + d(u, v)\). Then

\[
f(V_{-v}(u), u) = t' \leq t' + w = (t' + w + d(u, v)) - d(u, v) = \lim_{x \downarrow u} f(V_{-v}(u) \cup \{x\}, x).
\]

Thus condition (2) is also satisfied at \(x = u\). Note that \(w > 0\) only occurs if there is congestion at \((u, v)\) and this forces a left discontinuity, which is why the range in point (1) does not include \(x = u\).

The following lemma follows easily from the definitions.

**Lemma 19** Let \(T = (V, E)\) be a tree, \(e = (u, v) \in E\) and \(f(\cdot, \cdot)\) a minmax monotone cost function satisfying the continuity and non-decreasing properties of Definition 23. Then

\[
s_T = \max \left\{ x \in e : (f(V_{-v}(u) \cup \{x\}, x) \leq T \right\}
\]

(12)

(where \(\max\) denotes the closest point to \(v\) in the set) and

\[
a = \min_{x \in e} \max \left( f(V_{-v}(u) \cup \{x\}, x), f(V_{-u}(v) \cup \{x\}, x) \right)
\]

(13)

exist.

We further assume that \(s_T\) and \(a\) can be calculated using \(O(1)\) oracle calls, i.e., in \(O(t_A(n))\) time where \(n = |V|\). This is obviously true in the sink evacuation case because of the linearity of the functions as given by (11).

### 6.1 Extending Theorem 2 to the Continuous Case

Recall that Lemma 1 found \((u, v)\) such that \(f(V_{-v}(u), u) \leq T\) but \(f(V_{-v}(u) \cup \{v\}, v) > T\), and then placed a sink on \(u\). The motivating intuition was that the peaking condition implies that \(V_{-v}(u)\) MUST contain at least one sink. Placing that sink on the most extreme location possible for a single sink serving all of \(V_{-v}(u)\), i.e., \(u\), could only improve the sink assignment.

In the continuous case, the analogous argument is again that placing the sink on the most extreme location possible for serving \(V_{-v}(u)\) can only improve the sink assignment. But now, the most extreme location possible is no longer required to be \(u\); it is the unique point \(s_T\) defined in (12). (Fig. 19)

The Peaking Lemma for the continuous case will now create a new node at \(s = s_T\), splitting \((u, v)\) into two pieces. It will then place a sink on \(s\), committing all of \(V_{-v}(u)\) to \(s\) and adding \(s\) to \(S_{\text{out}}\). No changes need to be made to the Reaching Lemma, which will remain correct as stated. It can then be verified that the implementation of the peaking and reaching phases (including the first peaking phase via centroid decomposition) remain valid. Thus, the remainder of the bounded-cost minmax \(k\)-sink algorithm will follow exactly as it did before, with the running time remaining the same as well.
The hub tree after the sinks have been placed in the continuous problem. Five sinks $s_1, s_2, s_3, s_4, s_5$ have been placed by the peaking lemma. Note that the $s_i$ are not necessarily in $V$; they can be located somewhere on the edge $(u_i, v_i)$. In the feasibility version of the problem, the exact location of $s_i$ on the edge is known. In the minimization version, only the fact that $s_i$ falls in the edge $(u_i, v_i)$ is known but its exact location might not be.

### 6.2 Extending Theorem 3 to the Continuous Case

Let $B'$ be the new bounded cost minmax $k$-sink algorithm for the continuous case described in the previous subsection and $B'(n) = B(n)$ be the cost of running the algorithm on an input of size $n$. We now apply parametric search to $B'$ to create a general algorithm $I'$ for the continuous case. Some subtle differences between this and the application of parametric search to the bounded algorithm $B$ in Sect. 5 need to be noted.

Let $I'$ be the interfered (parametric search) version of $B'$ to be developed. Similar to $I$, $I'$ maintains a threshold range $(T^L, T^H)$. $I$ starts with $(T^L, T^H) = (0, +\infty)$ and maintains the same invariants:

1. $T^L < T^H$.
2. $T^L$ never decreases and $T^H$ never increases.
3. $T^L$ will be infeasible and $T^H$ will be feasible.
4. At each step of $I'$, the corresponding state of $B'$ would be identical for ALL values of $T \in [T^L, T^H]$.

The major difference will be in the definition of state and, in particular, what is stored in $S_{\text{out}}$. Recall that previously $S_{\text{out}} = \{s_1, \ldots, s_t\}$ was the set of known sinks (created by the peaking lemma).

As noted in Sect. 6.1, sink $s = s_T$ determined by the peaking lemma in the continuous case is no longer required to be a $u \in V$ but may lie inside an edge $(u, v)$. $B'$ explicitly determined the location of $s_T$ from $T$ using (12). In $I'$, $T$ is no longer exactly known, so (12) can no longer be applied.
To patch this, $S_{out}$ will no longer store the (unknown) location of sink $s$ but rather the directed edge $(u(s), v(s))$ which is known to contain $s$. (Fig. 19)

**Definition 24** The State of algorithm $B'$ at any given time will be $(S_{out}, P_{out})$ and the $S(v)$ values in the hub tree. $s \in S_{out}$ will be specified in the list by storing the edge $s = (u(s), v(s))$. (Fig. 19)

We now work through the differences between $I$ and $I'$

- **Algorithm 2**: The Group-Queries for the stages work almost exactly the same in $I'$ as in $I$. The only difference is that, after each Group-Query in a stage, $S_{out}$ will store the edge $(u(s), v(s))$ known to contain sink $s$, rather than the actual sink $s$ (which is unknown). The other edges are marked appropriately as in $I$.

- **Algorithm 3**: The Single-Queries in lines 16 and 19 and in Psearch in run exactly the same in $I$ as in $I'$. The only difference is that "Commit $V_{\bar{u}}(\bar{u})$ to $\bar{u}$" in lines 16 and 19 now add the edge $(\bar{u}, \bar{v})$ (which we know contains the sink) to $S_{out}$ instead of adding $\bar{u}$.

- **Algorithm 4**: Recall that the purpose of this algorithm is, for each $s \in ST$ to find the highest node $u_s$ on $\Pi(s, r)$ satisfying $f(BP(u_s, s), s) \leq T^*$. The Group-Queries for the stages work almost exactly the same in $I'$ as in $I$.

The only change is to replace the original contents of Algorithm 4, line 8,

8. Evaluate $a_s = f(BP(v_{m_s}^s, s), s)$

with a new line 8,

8. Evaluate $a_s := \min_{x \in (u(s), v(s))} \max \left( f(P_s \cup \{x\}, x), f(BP(v_{m_s}^s, x), x) \right)$,

where $(u(s), v(s))$ is the edge containing sink $s$.

It is not difficult to see that this modification correctly finds $u_s$ in the continuous case.

Lemma 17 and its proof will still work for this modified version with the new definition of state and the change in Algorithm 4, line 8 (Fig. 20).

**Lemma 20** Let $T^*$ be the optimal value of $T$ and let $[T^{L_i}, T^{H_i}]$ be the threshold range immediately before running the $i$th step (either a single or group step) of algorithm $I'$. Let $[T^<, T^>]$ be the threshold range when $I'$ terminates. Then

1. $\forall i, T^* \in [T^{L_i}, T^{H_i}]$.

2. Let $T' \in [T^{L_i}, T^{H_i}]$. Then, the state of Algorithm $B'$ with $T = T'$ immediately before running its $i$th step will be in exactly the same as the state of algorithm $B'$ with $T = T'$ immediately before running its $i$th step.

3. Algorithm $I'$ will end in exactly the same state as as algorithm $B'$ run with $T = T^<$. 

**Proof** (1) The proof is exactly the same as in Lemma 17.

(2) Again the proof is by induction on $i$, that after $m$ steps algorithm $B'$ will be in the same state when run on $T'$ and $T'$. Assume this is true before the $i$'th step and now consider what happens in the $i$'th step.
Fig. 20  An evaluation in line 8 of Algorithm 4 in the continuous case. A previous peaking step determined that edge \((u(s), v(s))\) contains sink \(s\). For a given ancestor \(v_{m_s}^a\) of \(v(s)\) in \(T\), line 9 finds the sink location \(x \in (u(s), v(s))\) that minimizes the maximum cost of servicing both \(P_s = V_{v(s)}(u(s))\) and \(BP(v_{m_s}^a, v(s))\).

In the diagram, the gray area is \(BP(v_{m_s}^a, x)\); it is the path from \(x\) to \(v_{m_s}^a\) and all of the outstanding branches that fall off of it. The unfilled nodes are known hub nodes. The filled nodes denote sink “locations”. The actual locations are unknown; only the (dashed) edges which contain them are known.

Fig. 21  Illustration of \(s_L \leq s'\) in the proof of Lemma 20. Recall that \(s_L\) is the rightmost location of a sink on edge \((u(s), v(s))\) that supports \(P_s\) when \(T = T^L\) and \(s'\) is the rightmost that supports it when \(T = T'\).

The analysis of the group-queries in Algorithm 2 and the single-queries in Algorithm 3 is exactly the same as it was in the proof Lemma 17, so we do not repeat it except to note again, that in all cases, the algorithm correctly processes “\(a \leq T^L\)”. Furthermore “\(a \leq T'\)” resolves identically to the query “\(a \leq T''\)” so \(B'\) can not distinguish between the two cases.

The analysis of the group-queries in Algorithm 4 is more subtle. Consider running the algorithm immediately after a group query has been run, and line 11, “If \(a_s \leq T'\)”, is being evaluated for some \(s\). We must show that, for any \(T' \in (T^L, T^H)\), the evaluation of “\(a_s \leq T'\)” returns the same answer as “\(a_s \leq T^L\)”.

Let \(e = (u(s), v(s))\) be the edge containing sink \(s\). Set \(s_L = s_{T^L}\) and \(s' = s_{T'}\) as defined in (12). By the induction hypothesis, both of these values are on the edge \(e\).

Since \(T^L < T'\), monotonicity implies \(s_L \leq s'\) (Fig. 21). Note that

\[
BP(v_{m_s}^a, s_L) = BP(v_{m_s}^a, v(s)) \cup \{s_L\} \quad \text{and} \quad BP(v_{m_s}^a, s') = BP(v_{m_s}^a, v(s)) \cup \{s'\}.
\]

Thus, from monotonicity,

\[
f(BP(v_{m_s}^a, s_L), s_L) \geq f(BP(v_{m_s}^a, s'), s').
\]
Recall that
\[ a_x = \min_{x \in (u(s), v(s))} \max \left( f(P_x \cup \{x\}, x), f(BP(v^x_{m_x}, x), x) \right). \]

Set \( x_s \in (u(s), v(s)) \) such that
\[ a_s = \max \left( f(P_x \cup \{x_s\}, x_s), f(BP(v^x_{m_x}, x_s), x_s) \right). \]

Because of the way that the group queries are defined in Line 4 of Figure 17 (with the resetting of \( T_L \) or \( T_H \)), we know that either \( a_s \leq T_L \) or \( a_s \geq T_H \).

We now separately analyze the two cases.

(i) \( a_s \leq T_L \):
Then \( f(P_x \cup \{x_s\}, x_s) \leq a_s \leq T_L \) implies \( x_s \leq s_L \leq s' \). From monotonicity and (14)
\[ T' > T_L \geq a_s = f(BP(v^x_{m_x}, x_s), x_s) \geq f(BP(v^x_{m_x}, s_L), s_L) \geq f(BP(v^x_{m_x}, s'), s'). \]
Then
\[ f(BP(v^x_{m_x}, s_L), s_L) \leq T_L \quad \text{and} \quad f(BP(v^x_{m_x}, s'), s') \leq T'. \] (15)

(ii) \( a_s \geq T_H \):
By the definition of \( a_s \),
\[ \max \left( f(P_x \cup \{s'\}, s'), f(BP(v_i, s'), s') \right) \geq a_s \geq T_H. \]

Since \( f(P_x \cup \{s'\}, s') \leq T' < T_H \), this and (14) imply
\[ f(BP(v^x_{m_x}, s_L), s_L) \geq f(BP(V, s'), s') \geq T_H > T' > T_L. \]

Then
\[ f(BP(v^x_{m_x}, s_L), s_L) > T_L \quad \text{and} \quad f(BP(v^x_{m_x}, s'), s') > T'. \] (16)

Combining (i) and (ii) show that the queries

"f(BP(V, s_L), s_L) \leq T_L?" \quad \text{and} \quad "f(BP(V, s'), s') \leq T'?"

will always return the same answer.

Since the algorithm started the group-query in an identical state for cases \( T = T', T_L \), and it can not distinguish between those cases after the group-query, it ends in the same state for both of them.

(3) follows from the analysis of (2).
We can now prove

**Lemma 21**

1. Let \((T^<, T^>)\) be the threshold range when \(I'\) terminates and \(T^*\) be the optimal value of \(T\). Then \(T^* = T^>\).
2. In particular, running the bounded cost Algorithm \(B'\) on \(T = T^>\) will retrieve the optimal feasible configuration.
3. Algorithm \(I'\) will terminate in \(T(n) = O(\max(k \log k, \log n)k^2nA(n))\) log\(^2\) time.

The proof of this lemma is almost exactly the same as that of Lemma 18 and will therefore be omitted. The only difference is that \(I'\) needs to do a bit of extra work for the Group-Queries in the reaching phase. But this is only \(O(t_A(n))\) work per sink and there are at most \(k\) sinks. The extra work is therefore \(O(kt_A(n))\), which is subsumed by the remaining running time of the algorithm, which in turn is the same as that for the discrete case.

### 7 The Fixed Sink Problem (Optimal Partitioning)

This section sketches a proof of Theorem 4, i.e., the special case in which the locations of \(S\), the set of \(k\) sinks in the input tree \(T_{in}\), are provided as part of the input. The problem is thus to partition \(T_{in}\) into \(k\) subtrees, each subtree containing exactly one \(s \in S\), so as to minimize the max-cost of the subtrees.

Because the sinks are given they can be considered as nodes in \(V\) and thus this problem is always discrete. Also, as stated in Theorem 4, the underlying function is now only required to be relaxed minmax monotone and not strictly minmax monotone. We explain below why this relaxation occurs.

#### 7.1 If \(S\) are All Leaves of \(T_{in}\)

Consider the special case in which all nodes in \(S\) are *leaves* of the input tree \(T_{in}\). As before, we start by constructing a feasibility test; given \(T > 0\), decide whether there exists a partition with bounded cost \(\leq T\).

As a first step, the new algorithm will build the hub tree \(T_{H(S)}\) (Def. 11) off of the given sinks. Let \(T_i, i = 1, 2, \ldots, t\), be the corresponding outstanding branches of \(T_{in}\) and \(w_i \in T_{H(S)}\) the node off of which \(T_i\) falls. Set

\[
T_{min} = \min_i f (T_i \cup \{w_i\}, w_i)
\]

By asymptotic subadditivity, calculating \(T_{min}\) requires only \(O(t_A(n))\) time.

If \(T_{min} > T\), no feasible solution exists. Otherwise, \(T_{H(S)}\) is RC-viable and, from the perspective of algorithm \(B\), the state of the problem is exactly the same as if the first peaking phase had just concluded with \(T = T_{in}\, S_{out} = S\) and \(P_{out}\) being defined by setting \(P_s = \{s\}\) for all \(s \in S_{out}\). Referring to Sect. 4.4, this is as if Step 1 of the algorithm had just concluded and the algorithm is now starting Step 2 (Creating the First Hub Tree). Continuing to run \(B\) from this point will now provide the correct answer. Since new sinks will never be added, the algorithm will never need to enter
Fig. 22  The original tree with specified sinks $s_1, s_2, s_3, s_4$ is on the left. The transformed forest is on the right. Note that a new sink has been created for each edge adjacent to a sink in the original tree.

the part of the peaking phase in which a new sink is added. That is, lines 15, 16 and 20, 21 in Algorithm 3, should never be reached. If they are, $T$ is infeasible. If they are not, the algorithm just keeps on peeling off sinks using the reaching phase.

The path-monotonicity requirement of minmax monotone functions stated in Sect. 2.2.1 was only used in the derivation of the peaking condition and in, Lemmas 7 and 8, for identifying the location of a peaking edge; the reaching condition itself only required set-monotonicity. Since a peaking phase adding sinks is never entered in this fixed-sink case, the algorithm remains correct even for relaxed minmax monotone functions.

The running time of this algorithm will be $O(nk + t_A(n))$ (to calculate $T_{\text{min}}$ and the structure of the first hub tree) plus the cost of running $B$ when skipping the first peaking phase, which is again $O(k^2 t_A(n) \log n)$. Technically, the algorithm could be simplified by noting that no sinks are ever added, but this would not improve the worst case running time.

Now consider solving the general minmax $k$-sink problem by applying parametric search as in the creation of $I$.

Again start the algorithm by building the hub tree and calculating $T_{\text{min}}$. Next, run the bounded cost fixed-sink algorithm with $T = T_{\text{min}}$. As previously noted, $T_{\text{min}} \leq T^*$. So, if $T_{\text{min}}$ is feasible $T^* = T_{\text{min}}$ and the algorithm concludes. The total work performed so far is $O(nk + t_A(n))$ plus $O(k^2 t_A(n) \log n)$ for calling the bounded algorithm.

If $T_{\text{min}}$ is not feasible, set $(T^L, T^H] = (T_{\text{min}}, \infty]$. The algorithm is now in the same state as $I$ would have been in if $I$ had just completed the first peaking phase. Continuing to run $I$ from this point onward will yield the final answer. This cost of running $I$, omitting the first peaking phase, is still $I'(k, n) = O\left(\max(k \log k, \log n) k^2 t_A(n) \log^2 n\right)$, which will then be the overall time complexity.

7.2 If $S$ are not Restricted to be Leaves of $T_{\text{in}}$

The subsection above solved the minmax $k$ fixed-sink problem in $I'(k, n)$ time when $S$ is restricted to being leaves of $T_{\text{in}}$.

Note that $I'(k, n)$ is non-decreasing in $k$. 

 Springer
The solution for general position $S$ uses a standard transformation of $T_{in}$ into a forest.

For every given sink $s \in S$, let $N(s)$ denote its set of neighbors and for every $u \in N(s)$ create a new sink node $s_u$ and edge $(u, s_u)$. Delete the original nodes in $S$. (Fig. 22). What remains is a forest $T_1, T_2, \ldots, T_r$ of trees in which each $T_i$ contains sinks $S_i$ at its leaves, where $|S_i| = k_i \leq k$ sinks. At most one new node is created for every edge in the original tree so the total number of vertices in the forest is $< 2n$. Furthermore, every partition on the forest corresponds in the natural way with a partition in the original tree such that the minmax-cost partition of the forest corresponds to a minmax-cost partition of the tree that has the same cost.

It is not difficult to see that

$$\min_{P \in \Lambda[S]} f(P, S) = \min_{1 \leq i \leq r} F_i$$

where $F_i = \min_{P \in \Lambda[S_i]} f(P, S_i)$. That is, we can separately find the optimal partition for each subtree and knit them together to construct an optimal partition for the original tree.

Thus, to solve the problem on the original tree it suffices to solve it on each of the trees $T_i$ individually. Let $n_i$ be the number of nodes in tree $T_i$. Recall that the statement of Theorem 4 assumed asymptotic subadditivity and that $t_A(2n) = O(t_A(n))$. Thus, the total cost is also at most

$$\sum_{i=1}^{r} I'(k_i, n_i) = O(I'(k, 2n - 1)) = O(I'(k, n))$$

and we are done.

Note that for the sink evacuation problem, plugging in the $O(n \log^2 n)$ oracle used previously, this leads to a $O\left(\max(k \log k, \log n) k^2 n \log^4 n\right)$ time algorithm for the partitioning problem, substantially improving upon the the $O(n(c \log n)^{k+1}) [28]$ and $O(n^2 k \log^2 n) [29]$ algorithms when $4 < k \ll \sqrt{n}$.

8 Conclusion

Given a tree $T = (V, E)$, we derived an algorithm for finding the locations of $k$ sinks that minimized the minmax Centered $k$ partitioning cost for any minmax monotone cost function. This formulation included the sink-evacuation problem on Dynamic Flow network trees as a special case.

The algorithm was developed in two parts. Sections 3 and 4 developed a feasibility test, i.e., for $T > 0$, an algorithm for finding a placement of $k$ sinks that permits partitioning the tree with cost $\leq T$ (or determining that such a placement does not exist). Section 5 showed how to apply parametric search to modify this test to find the minimum feasible $T^*$. Section 6 extended the algorithms to work in the continuous case (in which sinks can be placed on edges). Finally, Sect. 7 discussed how to specialize this to the case in which the $k$ sinks are known in advance.
Assuming an $t_A(n)$ time oracle for calculating the cost of the fixed 1-sink problem on trees, our main algorithm works in $O(\max(k \log k, \log n)k^2t_A(n) \log^2 n)$ time.

These were the first known polynomial time algorithms for these sink location problems. The obvious direction for improvement would be to try to develop algorithms whose running times, like the $O(n)$ one for unweighted $k$-center [18] and the $O(n \log n)$ ones [38] for the weighted $k$-center problem, are only dependent upon $n$ and not $k$. As noted earlier, the bottleneck to this generalization seems to be that, unlike in those previous tree-partitioning problems, the cost oracle $f(U, s)$ here is permitted to be a complicated of $U$ and $s$ that can not be decomposed into simpler parts.

Acknowledgements The work of both authors was partially supported by Hong Kong RGC CERG Grant 16208415. The authors would also like to thank Professor Tsunehiko Kameda and the anonymous referees for their comments and suggestions which identified an error in an earlier version of Sect. 3.5, and also helped improve the earlier drafts of this paper.

Declarations

Conflict of interest The authors have no relevant financial or non-financial interests to disclose.

References

1. Agarwal, P.K., Sharir, M.: Efficient algorithms for geometric optimization. ACM Comput. Surv. (CSUR) 30(4), 412–458 (1998)
2. Agasi, E., Becker, R.I., Perl, Y.: A shifting algorithm for constrained min-max partition on trees. Discret. Appl. Math. 45(1), 1–28 (1993)
3. Aronson, J.E.: A survey of dynamic network flows. Ann. Oper. Res. 20(1), 1–66 (1989)
4. Becker, R.I., Perl, Y.: Shifting algorithms for tree partitioning with general weighting functions. J. Algorithms 4(2), 101–120 (1983)
5. Becker, R.I., Perl, Y.: The shifting algorithm technique for the partitioning of trees. Discret. Appl. Math. 62(1–3), 15–34 (1995)
6. Becker, R.I., Perl, Y., Schach, S.R.: A shifting algorithm for min-max tree partitioning. J. ACM (JACM) 29(1), 58–67 (1982)
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’17, pp. 133–144. Springer (2017)
8. Bhattacharya, B., Kameda, T.: Improved algorithms for computing minmax regret sinks on dynamic path and tree networks. Theoret. Comput. Sci. 607, 411–425 (2015)
9. Chen, D., Golin, M.J.: Sink evacuation on trees with dynamic confluent flows. In: 27th International Symposium on Algorithms and Computation (ISAAC 2016). Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik (2016)
10. Chen, D.Z., Li, J., Wang, H.: Efficient algorithms for the one-dimensional k-center problem. Theoret. Comput. Sci. 592, 135–142 (2015)
11. Chen, J., Kleinberg, R.D., Lovász, L., Rajaraman, R., Sundaram, R., Vetta, A.: (Almost) Tight bounds and existence theorems for single-commodity confluent flows. J. ACM 54(4), 16-es (2007)
12. Chen, J., Rajaraman, R., Sundaram, R.: Meet and merge: approximation algorithms for confluent flows. J. Comput. Syst. Sci. 72(3), 468–489 (2006)
13. Cole, R.: Slowing down sorting networks to obtain faster sorting algorithms. J. ACM (JACM) 4(1), 200–208 (1978)
14. Dressler, D., Strehler, M.: Capacitated confluent flows: complexity and algorithms. In: 7th International Conference on Algorithms and Complexity (CIAC’10), pp. 347–358 (2010)
15. Fleischer, L., Skutella, M.: Quickest flows over time. SIAM J. Comput. 36(6), 1600–1630 (2007)
16. Fleischer, L., Tardos, E.: Efficient continuous-time dynamic network flow algorithms. Oper. Res. Lett. 23(3), 71–80 (1998)
17. Ford, L.R., Fulkerson, D.R.: Constructing maximal dynamic flows from static flows. Oper. Res. 6(3), 419–433 (1958)
18. Frederickson, G.N.: Parametric search and locating supply centers in trees. In: Proceedings of the Second Workshop on Algorithms and Data Structures (WADS’91), pp. 299–319. Springer (1991)
19. Garey, M.R., Johnson, D.S: Computers and intractability: a guide to the theory of NP-completeness. W.H. Freeman and Company, New York (1979)
20. Golin, M., Khodabande, H., Qin, B.: Non-approximability and polylogarithmic approximations of the single-sink unsplittable and confluent dynamic flow problems. In: Proceedings of the 27th International Symposium on Algorithms and Computation (ISAAC’16) (2017)
21. Higashikawa, Y., Golin, M.J., Katoh, N.: Minimax regret sink location problem in dynamic tree networks with uniform capacity. In: Proceedings of the 8th International Workshop on Algorithms and Computation (WALCOM’2014), pp. 125–137 (2014)
22. Higashikawa, Y.: Studies on the space exploration and the sink location under incomplete information towards applications to evacuation planning. PhD thesis, Kyoto University (2014)
23. Hoppe, B., Tardos, É.: The quickest transshipment problem. Math. Oper. Res. 25(1), 36–62 (2000)
24. Kamiyama, N., Katoh, N., Takizawa, A.: Theoretical and practical issues of evacuation planning in urban areas. In: The Eighth Hellenic European Research on Computer Mathematics and its Applications Conference (HERCMA2007), pp. 49–50 (2007)
25. Kariv, O., Hakimi, S.L.: An algorithmic approach to network location problems. i: The p-centers. SIAM J. Appl. Math. 37(3), 513–538 (1979)
26. Lari, I., Puerto, J., Ricca, F., Scozzari, A.: Algorithms for uniform centered partitions of trees. Electron. Notes Discrete Math. 55, 37–40 (2016)
27. Lari, I., Ricca, F., Puerto, J., Scozzari, A.: Partitioning a graph into connected components with fixed centers and optimizing cost-based objective functions or equipartition criteria. Networks 67(1), 69–81 (2015)
28. Mamada, S., Makino, K.: An evacuation problem in tree dynamic networks with multiple exits. In: Tatsuo, A., Shigeru, Y., Kazuhi, M. (eds.) Systems & Human Science-For Safety, Security, and Dependability; Selected Papers of the 1st International Symposium SSR2003, pp. 517–526. Elsevier B.V. (2005)
29. Mamada, S., Uno, T., Makino, K., Fujishige, S.: A tree partitioning problem arising from an evacuation problem in tree dynamic networks. J. Oper. Res. Soc. Jpn 48(3), 196–206 (2005)
30. 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. Discret. Appl. Math. 154(2387–2401), 251–264 (2006)
31. Megiddo, N.: Combinatorial optimization with rational objective functions. Math. Oper. Res. 4(4), 414–424 (1979)
32. Megiddo, N., Tamir, A.: New results on the complexity of p-centre problems. SIAM J. Comput. 12(4), 751–758 (1983)
33. Megiddo, N., Tamir, A., Zemel, E., Chandrasekaran, R.: An $O(n \log^2 n)$ algorithm for the k’th longest path in a tree with applications to location problems. SIAM J. Comput. 10(2), 328–337 (1981)
34. Pascoal, M.M.B., Captivo, M.E.V., Clímaco, J.C.N.: A comprehensive survey on the quickest path problem. Ann. Oper. Res. 147(1), 5–21 (2006)
35. Perl, Y., Vishkin, U.: Efficient implementation of a shifting algorithm. Discret. Appl. Math. 12(1), 71–80 (1985)
36. Shepherd, F.B., Vetta, A.: The inapproximability of maximum single-sink unsplittable, priority and confluent flow problems. Theor. Comput. 13(20), 1–25 (2017)
37. Skutella, M.: An introduction to network flows over time. In: William, C., László, L., Jens, V. (eds.) Research Trends in Combinatorial Optimization, pp. 451–482. Springer (2009)
38. Wang, H., Zhang, J.: An $O(n \log n)$-time algorithm for the $k$-center problem in trees. SIAM J. Comput. 50(2), 602–635 (2021)