

木構造の動的ネットワーク上の施設配置問題に対する $O(n \log^2 n)$ 時間アルゴリズム

間々田聡子* 宇野毅明† 牧野和久* 藤重悟‡

* 大阪大学大学院基礎工学研究科 † 国立情報学研究所 ‡ 京都大学数理解析研究所

概要: 動的ネットワークとは、各枝に移動時間と容量が与えられているネットワークである。ここで、各点に供給量がある木構造の動的ネットワークが与えられているとき、全ての供給量をできるだけ早く送り込むことができるような点 v を求める問題を考える。この問題は、木構造ネットワークにおける動的フローと施設配置問題を複合したもので、木構造ネットワークにおける 1-センター問題の動的フロー版として考えることができる。本研究では、動的に構造変更が可能な平衡 2 分木である区間木を用いた、 $O(n \log^2 n)$ 時間アルゴリズムを提案する（ただし、 n は点数である）。

An $O(n \log^2 n)$ Algorithm for the Optimal Sink Location Problem on Dynamic Tree Networks

Satoko MAMADA* Takeaki UNO† Kazuhisa MAKINO* Satoru FUJISHIGE‡

* Osaka University † National Institute of Informatics ‡ Kyoto University

Abstract: In this paper, we consider a sink location in a dynamic network which consists of a graph with capacities and transit times on its arcs. Given a dynamic network with initial supplies at vertices, the problem is to find a vertex v as a sink in the network such that we can send all the initial supplies to v as quick as possible. Motivated by evacuation plans, we study dynamic networks of tree structure. We present an $O(n \log^2 n)$ time algorithm for the sink location problem, where n is the number of vertices in the network.

1. Introduction

We consider dynamic networks that include transit times on arcs. Each arc a has the transit time $\tau(a)$ specifying the amount of time it takes for flow to travel from the tail to the head of a . In contrast to the classical *static* flows, flows in a dynamic network are called *dynamic*. In the setting, the capacity of an arc limits the rate of the flow into the arc at each time instance. Dynamic flow problems were introduced by Ford and Fulkerson [6] in late 1950s (see e.g. [5]). Since then, dynamic flows have been studied extensively. One of the main reasons is that dynamic flow problems arise in a number of applications such as traffic control, evacuation plans, production systems, communication networks, and financial flows (see the surveys by Aronson [2] and Powell, Jaillet, and Odoni [14]). For example, for building evacuation [7], vertices $v \in V$ model workplaces, hallways, stairwells, and so on, and arcs $a \in A$ model the connection link between the adjacent components of the building. For an arc $a = (v, w)$, the capacity $u(a)$ represents the number of people which can traverse the link corresponding to a per unit time, and $\tau(a)$ denotes the time it takes to traverse a from v to w .

This paper addresses the sink location problem in dynamic networks: given a dynamic network with the initial supplies at vertices, find a vertex, called a *sink*, such that the completion time to send all the initial supplies to

the sink is as small as possible. In the setting of building evacuation, for example, the problem models the location problem of an emergency exit together with the evacuation plan for it.

Our problem is a generalization of the following two problems. First, it can be regarded as a dynamic flow version of the 1-center problem [13]. In particular, if the capacities are sufficiently large, our problem represents the 1-center location problem. Secondly, our problem is an extension of the location problems based on flow (or connectivity) requirements in static networks, which have received much attention recently [1, 11, 16, 17].

We consider the sink location problem in dynamic *tree* networks. This is motivated by evacuation plans [7, 8], for example. In an emergency, everyone has to be evacuated fairly and without confusion, and hence it is natural to assume that the possible evacuation routes form a tree. It is known [12] that the problem can be solved in $O(n^2)$ time by using two-phase algorithm, where n denotes the number of vertices in a given network. We show that the problem is solvable in $O(n \log^2 n)$ time.

Our algorithm is based on a simple single-phase procedure, but uses a new technique for implicit representation which allows us to quickly update tables g i.e., sets of time intervals $[\theta_1, \theta_2)$ with their height $g(\theta_1)$ to perform three operations *Add-Table* (i.e., adding tables), *Shift-Table* (i.e., shifting a table), and *Ceil-Table* (i.e.,

ceiling a table by a prescribed capacity). We generalize interval trees (standard data structures for tables) by attaching additional parameters and show that using the data structures, we can efficiently handle the above-mentioned operations. Especially, we can merge tables g_i in $O((\sum_i d_i) \log^2(\sum_i d_i))$ time, where we say that *tables g_i are merged* if g_i 's are added into a single table g after shifting and ceiling tables are performed, and d_i denotes the number of intervals in g_i . This result implies an $O(n \log^2 n)$ time bound for the location problem. We mention that our data structures may be of independent interest and useful for some other problems which manage tables.

We remark that our location problem for general dynamic networks can be solved in polynomial time by solving the quickest transshipment problem n times. Here the quickest transshipment problem is to find a dynamic flow that zeroes all given supplies and demands within the minimum time, and is polynomially solvable by an algorithm of Hoppe and Tardos [9]. However, since their algorithm makes use of submodular function minimization [10, 15] as a subroutine, it requires polynomial time of high degree. As a corollary of our result, this paper shows that the quickest transshipment problem can be solved in $O(n \log^2 n)$ time if a given network is a tree and has a single sink.

The rest of the paper is organized as follows. The next section provides some preliminaries and fixes notation. Section 3 presents a simple single-phase algorithm for the sink location problem, and Section 4 describes and discusses our data structures. In Section 5, we analyze the complexity of our single-phase algorithm with our data structures. Finally, we give some conclusions in Section 6.

2. Definitions and Preliminaries

Let $T = (V, E)$ be a tree with a vertex set V and an edge set E . Let $\mathcal{N} = (T, c, \tau, b)$ be a dynamic flow network with the underlying undirected graph being the tree T , where $c : E \rightarrow \mathbf{R}_+$ is a capacity function representing the least upper bound for the rate of flow through each edge per unit time, $\tau : E \rightarrow \mathbf{R}_+$ a transit time function, and $b : V \rightarrow \mathbf{R}_+$ a supply function. Here, \mathbf{R}_+ denotes the set of all nonnegative reals and we assume the number of vertices in T is at least two.

This paper addresses the problem of finding a sink $t \in V$ such that we can send given initial supplies $b(v)$ ($v \in V \setminus \{t\}$) to sink t as quick as possible. Suppose that we are given a sink t in T . Then, T is regarded as an in-tree with root t , i.e., each edge of T is oriented toward the root t . Such an oriented tree with root t is denoted by $\vec{T}(t) = (V, \vec{E}(t))$. Each oriented edge in $\vec{E}(t)$ is denoted by the ordered pair of its end vertices and is called an arc. For each edge $\{u, v\} \in E$, we write $c(u, v)$ and $\tau(u, v)$ instead of $c(\{u, v\})$ and $\tau(\{u, v\})$, respectively.

For any arc $e \in \vec{E}(t)$ and any $\theta \in \mathbf{R}_+$, we denote by $f_e(\theta)$ the flow rate entering the arc e at time θ which arrives at the head of e at time $\theta + \tau(e)$. We call $f_e(\theta)$ ($e \in \vec{E}(t)$, $\theta \in \mathbf{R}_+$) a *continuous-time dynamic flow* in $\vec{T}(v^*)$ (with a sink v^*) if it satisfies the following three conditions, where $\delta^+(v)$ and $\delta^-(v)$ denote the set of all arcs leaving v and entering v , respectively.

- (a) (Capacity constraints): For any arc $e \in \vec{E}(t)$ and $\theta \in \mathbf{R}_+$,

$$0 \leq f_e(\theta) \leq c(e).$$

- (b) (Flow conservation): For any $v \in V \setminus \{v^*\}$ and $\Theta \in \mathbf{R}_+$,

$$\sum_{e \in \delta^+(v)} \int_0^\Theta f_e(\theta) d\theta - \sum_{e \in \delta^-(v)} \int_{\tau(e)}^\Theta f_e(\theta - \tau(e)) d\theta \leq b(v).$$

- (c) (Demand constraints): There exists a time $\Theta \in \mathbf{R}_+$ such that

$$\begin{aligned} \sum_{e \in \delta^-(v^*)} \int_{\tau(e)}^\Theta f_e(\theta - \tau(e)) d\theta - \sum_{e \in \delta^+(v^*)} \int_0^\Theta f_e(\theta) d\theta \\ = \sum_{v \in V \setminus \{v^*\}} b(v). \end{aligned} \quad (1)$$

As seen in (b), we allow intermediate storage (or holding inventory) at each vertex. For a continuous-time dynamic flow f , let θ_f be the minimum time θ satisfying (1), which is called the *completion time* for f . We further denote by $C(v^*)$ the minimum θ_f among all continuous dynamic flows f in $\vec{T}(v^*)$. We study the problem of computing a sink $v^* \in V$ with the minimum $C(v^*)$. This problem can be regarded as a dynamic version of the 1-center location problem (for a tree) [13]. In particular, if $c(v, w) = +\infty$ (a sufficiently large real) for each edge $\{v, w\} \in E$, our problem represents the 1-center location problem [13].

We remark that dynamic flows can be restricted to those having no intermediate storage without changing optimal sinks of our problem (see discussions in [6, 9, 12], for example).

2.1. An $O(n^2)$ algorithm given in [12]

In this section, we review the outline of an $O(n^2)$ algorithm which has been proposed in [12], in order to make our faster algorithm easily understood.

The algorithm consists of two phases, Phases I and II. Phase I arbitrarily chooses a vertex $t \in V$ as a candidate sink and compute the completion time $C(t)$ and a dynamic flow f that completes in $C(t)$. Then Phase II computes an optimal sink t^* by repeatedly picking up a new candidate sink \hat{t} that is adjacent to the current one t and updating $t := \hat{t}$ if $C(\hat{t}) < C(t)$.

In both phases, we keep two tables, *Arriving Table* A_v and *Sending Table* S_v for each vertex $v \in V$. Arriving Table A_v represents the sum of the flow

rates arriving at vertex v as a function of time θ , i.e., $\sum_{e \in \vec{E}(t): e=(u,v)} f_e(\theta - \tau(e)) + \eta_\theta(v)$, where $f_e(\theta) = 0$ holds for any $e \in \vec{E}(t)$ and $\theta < 0$, and $\eta_\theta(v) = \frac{b(v)}{\Delta}$ if $0 \leq \theta < \Delta$; otherwise 0. Here, Δ denotes a sufficiently small positive constant. Sending Table S_v represents the flow rate leaving vertex v as a function of time θ , i.e., $f_{(v,w)}(\theta)$, where $(v,w) \in \vec{E}(t)$.

Let us consider a table $g : \mathbf{R}_+ \rightarrow \mathbf{R}_+$, which represents the flow rate in time $\theta \in \mathbf{R}_+$. Here, we assume $g(\theta) = 0$ for $\theta < 0$. Since our problem can be solved by sending out as much amount of flow as possible from each vertex to its parent if a candidate sink t is chosen in advance, we only consider the table g which is representable as

$$g(\theta) = \begin{cases} 0 & \text{if } \theta < \theta_1 \\ g(\theta_i) & \text{if } \theta_i \leq \theta < \theta_{i+1} \text{ for } i = 1, \dots, k-1 \\ 0 & \text{if } \theta \geq \theta_k, \end{cases}$$

where $\theta_i < \theta_{i+1}$ and $g(\theta_i) \neq g(\theta_{i+1})$ for $i = 1, \dots, k$. Thus, we represent such tables g by a set of intervals (with their height), i.e.,

$$((-\infty, \theta_1), 0), ([\theta_i, \theta_{i+1}), g(\theta_i)) \quad (i = 1, \dots, k),$$

where $\theta_{k+1} = +\infty$ and $g(\theta_k) = 0$. A time θ is called a *jump time* of g if $\lim_{x \rightarrow -0} g(\theta + x) \neq \lim_{x \rightarrow +0} g(\theta + x)$.

Let us now describe Phases I and II as follows.

Algorithm DOUBLE-PHASE

(Phase I)

Step 0: Choose a vertex t arbitrarily. Put $T' \leftarrow \vec{T}(t)$.

Step 1: If T' consists of t alone, then go to Step 3. For each leaf v of T' , construct Sending Table S_v from Arriving Table A_v by bounding A_v by $c(v,w)$, where w is a parent of v in T' .

Step 2: For each non-leaf w whose children are all leaves, construct Arriving Table A_w from Sending Tables S_v of its children v by shifting A_v right by $\tau(v,w)$ and adding all such shifted tables and the initial supply $\eta_\theta(w)$.

Remove all the leaves $v (\neq t)$ from T' and denote the resultant tree by T' again.

Go to Step 1.

Step 3: Compute the completion time $C(t)$ from A_t .

(Phase II)

Step 0: Find a child v of root t that sends t the last flow (i.e., the flow that arrives at time $C(t)$). Put $\hat{t} \leftarrow v$ and consider \hat{t} as a new sink.

Step 1: Compute the completion time $C(\hat{t})$ and the corresponding tables as follows.

(1-1) Compute new Arriving Table \tilde{A}_t by subtracting from A_t the table obtained from $S_{\hat{t}}$ by shifting it right by $\tau(\hat{t}, t)$.

(1-2) Compute from \tilde{A}_t Sending Table S_t to go through (t, \hat{t}) (as in Step 1 of Phase I).

(1-3) Compute Arriving Table $\tilde{A}_{\hat{t}}$ by adding $A_{\hat{t}}$ and the table constructed from S_t by shifting it right by $\tau(t, \hat{t})$. Compute the completion time $C(\hat{t})$.

Step 2:

(2-1) If $C(t) < C(\hat{t})$, then return $t^* = t$ and halt.

(2-2) If $C(t) \geq C(\hat{t})$ and the last flow reaches sink \hat{t} from t , then return $t^* = \hat{t}$ and halt.

(2-3) Otherwise, put $t \leftarrow \hat{t}$ and go to Step 0. \square

Note that tables A_v and S_v can be constructed by adding, shifting, and/or bounding the other tables. Now, we more formally describe how to compute them.

In Step 1 of Phase I, Arriving Table A_v for a leaf v of the original $\vec{T}(t)$ is given as

$$((-\infty, 0), 0), ([0, \Delta), b(v)/\Delta), ([\Delta, +\infty), 0), \quad (2)$$

and Sending Table S_v for a leaf v of T' can be constructed from A_v as follows. Let A_v be represented as

$$((-\infty, \theta_1), 0), ([\theta_i, \theta_{i+1}), h_i) \quad (i = 1, 2, \dots, k),$$

where $\theta_{k+1} = +\infty$ and $h_k = 0$, and let $R_i = (h_i - c(v,w))(\theta_{i+1} - \theta_i)$.

Step 1: Output $((-\infty, \theta_1), 0)$ and $i := 1$

Step 2: If $R_i < 0$, then output $([\theta_i, \theta_{i+1}), h_i)$, and $i := i + 1$. Otherwise, let α be an integer such that $\sum_{\ell=i}^j R_\ell \geq 0$ for $j \leq \alpha - 1$ and $\sum_{\ell=i}^\alpha R_\ell < 0$ and let $\beta = \theta_\alpha + \sum_{\ell=i}^{\alpha-1} R_\ell / (c(v,w) - h_\alpha)$. Then output $([\theta_i, \beta), c(v,w))$ and $([\beta, \theta_{\alpha+1}), h_\alpha)$, and $i := \alpha + 1$.

Step 3: If $i = k + 1$, then halt. Otherwise, go to Step 2.

Step 2 of Phase I computes Arriving Table A_w from S_v for children v 's of w and the initial supply of w as follows.

For a child v of w , let S_v be represented as $((-\infty, \theta_1^v), 0), ([\theta_i^v, \theta_{i+1}^v), h_i^v) \quad (i = 1, 2, \dots, k_v)$, where $\theta_{k_v+1}^v = +\infty$ and $h_{k_v}^v = 0$, and let the initial supply of w be represented as in (2): $((-\infty, 0), 0), ([0, b(w)/\Delta), \Delta), ([b(w)/\Delta, +\infty), 0)$. From these tables, we first sort all the elements in $\{\theta_i^v + \tau(v,w) \mid v : \text{a child of } w, i = 1, \dots, k_v + 1\} \cup \{0, b(w)/\Delta, +\infty\}$ as $\theta_1 < \theta_2 < \dots < \theta_{k+1} (= +\infty)$, and then output $((-\infty, \theta_1), 0)$ and

$$\left([\theta_i, \theta_{i+1}), \sum_{v: \text{a child of } w} h^v(\theta_i - \tau(v,w)) + h^w(\theta_i) \right),$$

where $i = 1, 2, \dots, k$, $h^v(\theta)$ and $h^w(\theta)$ denote the height of the table S_v and the initial supply of w at time θ , respectively.

By using similar methods, Phase II computes the tables.

It was shown in [12] that Algorithm DOUBLE-PHASE correctly computes an optimal sink and it requires $O(n^2)$ time. The latter follows from the fact that each table g can be computed in time linear in the total number of intervals in the tables from which g is constructed and the number of intervals in each table is linear in n .¹ Namely, we have the following theorem.

¹ It was shown in [12] that the number of intervals is at most $3n$ for discrete-time dynamic flows.

Theorem 2.1 ([12]): *Algorithm DOUBLE-PHASE solves the sink location problem in $O(n^2)$ time.* \square

3. A Single-Phase Algorithm

Algorithm DOUBLE-PHASE consists of two phases. This section presents a simple $O(n^2)$ algorithm with a single phase. Because of the simplicity, it gives us a good prospect to develop a faster algorithm. In fact, we can construct an $\tilde{O}(n)$ algorithm based on this framework, which is given in the next section.

Intuitively, our single-phase algorithm first constructs Arriving Tables A_v for all leaves v . Then the algorithm finds a leaf v^* such that T has an optimal sink other than v^* and removes it from T . If some vertex v becomes a leaf of the modified tree T , then the algorithm computes Arriving Table A_v for such a new leaf v by using Arriving tables for the vertices that are adjacent to v and have already been removed. The algorithm repeatedly applies this procedure to T until T becomes a single vertex t , and outputs such a vertex t as an optimal sink.

Algorithm SINGLE-PHASE

Step 0: Let $W := V$, and let L be the set of all leaves of T .

For each $v \in L$, construct Arriving Table A_v .

Step 1: For each $v \in L$, construct from A_v Sending Table S_v to go through $(v, p(v))$, where $p(v)$ is a vertex adjacent to v in T . Compute the time $Time(v, p(v))$ at which the flow based on S_v is completely sent to $p(v)$.

Step 2: Compute a vertex $v^* \in L$ such that $Time(v^*, p(v^*)) = \min_{v \in L} Time(v, p(v))$. Let $W := W \setminus \{v^*\}$ and $L := L \setminus \{v^*\}$.

If there exists a leaf v of $T[W]$ such that v is not contained in L ,

then:

(1) Let $L := L \cup \{v\}$.

(2) Construct Arriving Table A_v from the initial supply $\eta_\theta(v)$ and Sending Table $S_{v'}$ for the vertices v' that are adjacent to v in T and have already been removed from W .

(3) Compute from A_v Sending Table S_v to go through $(v, p(v))$ where $p(v)$ is a vertex adjacent to v in $T[W]$, and compute $Time(v, p(v))$.

Step 3: If $|W| = 1$, then output $t \in W$ as an optimal sink. Otherwise, return to Step 2. \square

Here $T[W]$ denotes a subtree of T induced by a vertex set W , and tables A_v and S_v are constructed as in Algorithm DOUBLE-PHASE.

Note that at most one leaf v of $T[W]$ is not contained in L in the if-statement of Step 2, and L is always the set of all leaves of $T[W]$ before executing Step 2 in each iteration. Denote by $T_{(v,w)}$ the tree containing v and obtained by removing edge (v, w) from T , and by $T_{(v,w)}^+$ the trees obtained by adding $T_{(v,w)}$ to edge (v, w) . Then we can see that $Time(v, p(v))$ in Step 1 or 2 represents the completion time for $\overrightarrow{T_{(v,p(v))}^+}(p(v))$.

Lemma 3.1: *Algorithm SINGLE-PHASE outputs an optimal sink t .* \square

Similarly as Algorithm DOUBLE-PHASE, it is not difficult to see that Algorithm SINGLE-PHASE requires $\Theta(n^2)$ time if we construct Arriving and Sending Tables explicitly. In Section 4, we present a method to represent these tables implicitly, and develop an $O(n \log^2 n)$ time algorithm for our location problem.

4. Data Structure for Arriving and Sending Tables

This section presents sophisticated data structures which can be used to represent Arriving/Sending Tables *implicitly*. We adopt interval trees for them, which are standard data structures for a set of intervals. Note that SINGLE-PHASE only applies to tables A_v and/or S_v the following three basic operations: *Add-Table* (i.e., adding tables), *Shift-Table* (i.e., shifting a table), and *Ceil-Table* (i.e., ceiling a table by a prescribed capacity). It is known that interval trees can efficiently handle operations *Add-Table* and *Shift-Table* (see Section 4.1). However, standard interval trees cannot efficiently handle operation *Ceil-Table*. This paper develops new interval trees which efficiently handle all the three operations.

4.1. Interval Trees

This section explains our interval trees for data structures of tables A_v and S_v . Our interval trees are implemented by using binary balanced trees. Let g be a table represented as

$$I_i = ([\theta_i, \theta_{i+1}), g(\theta_i)) \quad (i = 0, 1, \dots, k), \quad (3)$$

where $\theta_0 = -\infty$, $\theta_{k+1} = +\infty$, and $g(\theta_0) = g(\theta_k) = 0$,² and let BT_g denote a binary tree for g . We denote the root by r^{BT} and the height of BT by $height(BT)$. The binary tree BT_g has an additional parameter t_{base} to represent how much g is shifted right. This t_{base} is used for operation *Shift-Table* by updating t_{base} to $t_{base} + \mu$, where μ denotes the time to shift the table right. Moreover, each node x in BT_g has five nonnegative parameters $base(x)$, $ceil(x)$, $h_e(x)$, $t^r(x)$, and $t^l(x)$ with $t^l(x) \leq t^r(x)$, and each leaf has $e(x)$ in addition, where these parameters will be explained later. A leaf x is called *active* if $t^l(x) < t^r(x)$ and *dummy* otherwise. The time intervals of a table g correspond to the active leaves of BT_g bijectively. We denote by $\#(BT)$ the number of active leaves of BT .

Initially (i.e., immediately after constructing BT_g by operation MAKETREE given below), BT_g contains no dummy leaf and hence there exists a one-to-one correspondence between the time intervals of g and leaves of BT_g . Moreover, for each leaf x corresponding to I_i in (3), we have $t^l(x) = \theta_i$, $t^r(x) = \theta_{i+1}$, $base(x) = g(\theta_i)$ and $ceil(x) = +\infty$, and for each non-leaf x , $t^l(x) = \min_{y \in Leaf(x)} t^l(y)$, $t^r(x) = \max_{y \in Leaf(x)} t^r(y)$,

² For simplicity, we write the first interval I_0 as $([-\infty, \theta_1), 0)$ instead of $((-\infty, \theta_1), 0)$.

$base(x) = 0$ and $ceil(x) = +\infty$. Here, $Leaf(x)$ denotes the set of all leaves which are descendants of x . Namely, $t^l(x)$ and $t^r(x)$, respectively, represent the start and the end points of the interval corresponding to x , and $base(x)$ and $ceil(x)$, respectively, represent the flow rate and the upper bound for the flow rate in the time interval corresponding to x .

Operation MAKETREE (g : table)

Step 1: Let $t_{base} := 0$.

Step 2: Construct a binary balanced tree BT_g whose leaves x_i correspond to the time interval I_i of g in such a way that the leftmost leaf corresponds to the first interval I_0 , the next one corresponds to the second interval I_1 , and so on.

Step 3: For each leaf x_i corresponding to interval $I_i = [\theta_i, \theta_{i+1})$, $base(x) := g(\theta_i)$, $t^l(x) := \theta_i$ and $t^r(x) := \theta_{i+1}$.

Step 4: For each non-leaf x , $base(x) := 0$, and $t^l(x) := \min_{y \in Leaf(x)} t^l(y)$ and $t^r(x) := \max_{y \in Leaf(x)} t^r(y)$.

Step 5: For each node x , $ceil(x) := +\infty$.

Step 6: For each leaf x , set $e(x)$, and for each node x , set $h_e(x)$, where $e(x)$ and $h_e(x)$ shall be explained later. \square

We can easily compute a table g from BT_g constructed by MAKETREE. It should also be noted that a binary tree BT_g is not unique, i.e., distinct trees may represent the same table g .

As mentioned in this section, *Shift-Table* can easily be handled by updating t_{base} . We now consider *Add-Table*, i.e., constructing a table g by adding two tables g_1 and g_2 , where we regard an addition of k tables as $k - 1$ successive additions of two tables. Let us assume that $\#(BT_{g_1}) \geq \#(BT_{g_2})$, that is, g_1 has at least as many intervals as g_2 . Our algorithm constructs BT_g by adding all intervals (corresponding to active leaves) of BT_{g_2} one by one to BT_{g_1} . Each addition of an interval $([\theta_1, \theta_2), c)$ to BT_{g_1} , denoted by $\text{ADD}(BT_{g_1}; \theta_1, \theta_2, c)$, can be performed as follows.

We first modify BT_{g_1} to \widetilde{BT}_{g_1} that has (active) leaves x and y such that $t^l(x) = \theta_1$ and $t^r(y) = \theta_2$ if there exist no such leaves. We can see that the time interval $[\theta_1, \theta_2)$ can be represented by the union of disjoint maximal intervals in \widetilde{BT}_{g_1} , i.e., the set of incomparable nodes in \widetilde{BT}_{g_1} , denoted by $rep(\theta_1, \theta_2)$. We then update $base$ of \widetilde{BT}_{g_1} as follows

$$base(x) := base(x) + c \text{ for all } x \in rep(\theta_1, \theta_2).$$

By successively applying this procedure to new interval tree \widetilde{BT}_{g_1} and each of the remaining intervals in BT_{g_2} , we can construct BT_g with $g = g_1 + g_2$.

For an interval tree BT and an active leaf x of BT , let $y_1(= x)$, $y_2, \dots, y_s(= r^{BT})$ denote the path from x to the root r^{BT} . The procedure given above shows that the height of an active leaf x representing the flow rate of the corresponding interval can be represented as

$$h(x) = \sum_{i=1}^s base(y_i). \quad (4)$$

Operation $\text{ADD}(BT_{g_1}; \theta_1, \theta_2, c)$ can be handled in $O(\log n)$ time, since $|rep(\theta_1, \theta_2)| \leq 2height(BT_{g_1}) = O(\log n)$. This means that BT_g can be constructed from BT_{g_1} and BT_{g_2} in $O(\#(BT_{g_2}) \log n)$ time. Moreover, operations *Add-Table* in Algorithm SINGLE-PHASE can be performed in $O(n \log^2 n)$ time in total, since we always add a smaller table to a larger one (see Section 4.3 for the details). Thus *Add-Table* can be performed efficiently.

However, operations *Ceil - Table* in Algorithm SINGLE-PHASE require $\Theta(n^2)$ time in total, since the algorithm contains $\Theta(n)$ *Ceil-Table*, each of which requires $\Theta(n)$ time, even if we use interval trees as data structures for tables. Therefore, when we bound BT by a constant c , we omit modifying t^l , t^r , and $base$, and keep c as $ceil(r^{BT}) = c$. Clearly, this causes difficulties to overcome as follows.

First, $h(x)$ in (4) does not represent the actual height any longer. We call $h(x)$ the *tentative height* of x in BT , and the *actual height* $\hat{h}(x)$ is computed as

$$\hat{h}(x) = h(x) - \max_k \left\{ 0, \sum_{i=1}^k base(y_i) - ceil(y_k) \right\}, \quad (5)$$

where $y_1(= x)$, $y_2, \dots, y_s(= r^{BT})$ denotes the path from x to r^{BT} . Note that $ceil(y_k)$ has finite value, since tables are added successively. Intuitively, for a node y_k in BT , $ceil(y_k)$ represents the upper bound of the height of active leaves $x \in Leaf(y_k)$ within the subtree of BT whose root is y_k . Thus $\sum_{i=1}^k base(y_i) - ceil(y_k)$ has to be subtracted from the height $h(x)$ if $\sum_{i=1}^k base(y_i) - ceil(y_k) > 0$, and the actual height $\hat{h}(x)$ is obtained by subtracting their maximum. Note that $\hat{h}(x) = h(x)$ holds for all active leaves x of a tree constructed by MAKETREE.

We next note that there exists no one-to-one correspondence between active leaves in BT and time intervals of the table that BT represents, if we just set $ceil(r^{BT}) = c$. In this case, the table is updated too drastically to efficiently handle the operations afterwards. Thus by modifying BT (as shown in the subsequent subsections), we always keep the one-to-one correspondence, i.e., the property that any two consecutive active leaves x and x' satisfy

$$\hat{h}(x) \neq \hat{h}(x'). \quad (6)$$

We finally note that, for an active leaf x , $t^l(x)$ and $t^r(x)$ do not represent the start and the end points of the corresponding interval. Let x be an active leaf in BT that does not correspond to the first interval or the last interval. For such an x , let x^- and x^+ denote active leaves in BT which are left-hand and right-hand neighbors of x , respectively, i.e.,

$$t^r(x^-) = t^l(x), \quad t^l(x^+) = t^r(x).$$

Then the start and the end points of the corresponding interval can be obtained by

$$\begin{aligned}\hat{t}^r(x) &= t_{base} + t^r(x) + \\ &\quad (t^r(x) - t^l(x)) \times \frac{h(x) - \hat{h}(x)}{\hat{h}(x) - \hat{h}(x^+)}\end{aligned}\quad (7)$$

$$\hat{t}^l(x) = \hat{t}^r(x^-) \quad (8)$$

Here $\hat{t}^r(x)$ and $\hat{t}^l(x)$ are well-defined from (6). For active leaves x and y corresponding to the first interval and the last interval, we have $\hat{t}^l(x) = -\infty$, $\hat{t}^r(x) = t^l(x^+)$, $\hat{t}^l(y) = \hat{t}^r(y)$ and $\hat{t}^r(y) = +\infty$.

It follows from (5), (7), and (8) that $\hat{h}(x)$, $\hat{t}^r(x)$, and $\hat{t}^l(x)$ can be computed from $base$, $ceil$, $t^r(x)$, and $t^l(x)$ in $O(\text{height}(BT))$ time. In order to check (6) efficiently, each active leaf x has

$$e(x) = \begin{cases} \max\{0, h(x) - h(x^+)\} \times \frac{t^r(x^+) - t^r(x)}{t^r(x^+) - t^l(x)} & \text{if } x^+ \text{ exists} \\ +\infty & \text{otherwise} \end{cases}$$

and each node x has

$$h_e(x) = \max_{y \in \text{Leaf}_A(x)} \left\{ \sum_{z \in \text{path}(x,y)} base(z) - e(y) \right\},$$

where $\text{Leaf}_A(x)$ denotes the set of active leaves that are descendants of x , and $\text{path}(x, y)$ denotes the set of nodes on the path from x to y . We have the following lemma.

Lemma 4.1: *Let BT be a binary tree in which $\hat{h}(x) \neq \hat{h}(x^+)$ holds for every active leaf x . After bounding BT by a constant c ,*

- (i) $\hat{h}(x) \neq \hat{h}(x^+)$ holds for an active leaf x if and only if x satisfies $h(x) - e(x) < c$,
- (ii) all active leaves x in BT satisfy $\hat{h}(x) \neq \hat{h}(x^+)$ if and only if $h_e(r^{BT}) < c$.

Moreover, we can compute an active leaf x with $\hat{h}(x) = \hat{h}(x^+)$ in $O(\text{height}(BT))$ time by scanning $h_e(x)$ from the root r^{BT} . Note that $h_e(x)$ can be obtained by the following bottom-up computation.

$$h_e(x) = \begin{cases} base(x) - e(x) & \text{if } x \text{ is a leaf} \\ \max\{h_e(x_1), h_e(x_2)\} + base(x) & \text{otherwise,} \end{cases} \quad (9)$$

where x_1 and x_2 denote the children of x . This means that preparing and updating h_e 's can be handled efficiently.

In summary, we always keep the following conditions for binary trees BT_g to represent tables g . Note that BT satisfies the conditions.

- (C0) For any node x , BT maintains $t^l(x)$, $t^r(x)$, $ceil(x)$, $base(x)$, and $h_e(x)$. For any leaf x , BT maintains $e(x)$ in addition.

- (C1) Any node x satisfies $t^l(x) \leq t^r(x)$. Any non-leaf x satisfies $t^l(x) = \min_{y \in \text{Leaf}(x)} t^l(y)$, and $t^r(x) = \max_{y \in \text{Leaf}(x)} t^r(y)$.

- (C2) Any active leaf x satisfies $t^r(x) = t^l(x^+)$.

- (C3) Any active leaf x satisfies $\hat{h}(x) \neq \hat{h}(x^+)$,

- (C4) Any active leaf x satisfies $\hat{h}(x) \geq h(x) - e(x)$.

A binary tree BT is called *valid* if it satisfies conditions (C0)~(C4). For example, a binary tree BT constructed by `MAKETREE` is valid.

4.2. Operation NORMALIZE

As discussed in Section 4.1, we represent a table g as a valid binary balanced tree BT . For an active leaf x , our algorithm sometimes need to update BT to get one having *accurate* x , i.e., $base$ and $ceil$ are updated so that

$$\begin{aligned} base(y) &:= \begin{cases} 0 & \text{for a proper ancestor } y \text{ of } x^- \text{ or } x \\ \hat{h}(y) & \text{for } y = x^- \text{ or } x \end{cases} \\ ceil(y) &:= +\infty \text{ for an ancestor } y \text{ of } x^- \text{ or } x \\ t^r(y) = t^l(y^+) &:= \hat{t}^r(y) \text{ for } y = x^- \text{ or } x \end{aligned} \quad (10)$$

For example, this can be performed before modifying BT to keep (6).

The following operation, called `NORMALIZE`, updates BT as above, and also maintains the balance of BT (i.e., $\text{height}(BT) = O(\log n)$).

Operation `NORMALIZE`(BT, x : an active leaf)

Step 1: Update $base$ and $ceil$ by the following top-down computation along the path from r^{BT} to the parent of y for $y = x^-$ or x . For a node z on the path and its children z_1 and z_2 ,

$$\begin{aligned} base(z_i) &:= base(z_i) + base(z), \quad base(z) := 0, \\ ceil(z_i) &:= \min\{ceil(z_i) + base(z), ceil(z)\}, \\ ceil(z) &:= +\infty. \end{aligned}$$

Step 2: If x was added to BT immediately before this operation, then rotate BT in order to keep the balance of BT .

Step 3: For $y = x, x^-$, if $base(y) > ceil(y)$, then $t^r(y) = t^l(y^+) := \hat{t}^r(y)$, $base(y) := ceil(y)$, otherwise $ceil(y) := +\infty$.

Step 4: For $y = x^-, x, x^+$, update t^l, t^r, e , and h_e by the bottom-up computation along the path from y to r^{BT} . \square

Note that nodes may be added to BT (by operation `SPLIT` in the next section), but are never removed from BT , although some nodes become dummy. This simplifies the analysis of the algorithm, since removing a node from BT requires the rotation of BT that is not easily implemented.

It is not difficult to see that the tree BT' obtained by `NORMALIZE` is valid, satisfies (10), and represents the

same table as BT . Moreover, since the lengths of the paths in Steps 1 and 4 are $O(\text{height}(BT))$, BT' can be computed from BT in $O(\text{height}(BT))$ time. Thus we have the following lemma.

Lemma 4.2: *Let BT be a valid binary balanced tree representing a table g , and let x be an active leaf of BT . Then BT' obtained by $\text{NORMALIZE}(BT, x)$ is a valid binary balanced tree that represents g and satisfies (10). Furthermore, BT' is computable from BT in $O(\text{height}(BT))$ time.*

4.3. Add-Table

This section shows how to add two binary balanced trees BT_{g_1} and BT_{g_2} for tables g_1 and g_2 . We have already mentioned an idea of our Add-Table after describing operation MAKETREE . Formally it can be written as follows.

Input: Two valid binary balanced trees BT_{g_1} and BT_{g_2} for tables g_1 and g_2 .

Output: A valid binary balanced tree BT_g for $g = g_1 + g_2$.

Step 1: If $\#(BT_{g_1}) \geq \#(BT_{g_2})$, then $BT_1 := BT_{g_1}$ and $BT_2 := BT_{g_2}$. Otherwise $BT_1 := BT_{g_2}$ and $BT_2 := BT_{g_1}$.

Step 2: For each active leaf $x \in BT_2$, compute $\hat{t}^l(x)$, $\hat{t}^r(x)$ and $\hat{h}(x)$, and call operation ADD for BT_1 , $\hat{t}^l(x)$, $\hat{t}^r(x)$, and $\hat{h}(x)$. \square

Operation $\text{ADD}(BT, \theta_1, \theta_2, c)$

Step 1: Call $\text{SPLIT}(BT, \theta_1 - t_{base}^{BT})$ and $\text{SPLIT}(BT, \theta_2 - t_{base}^{BT})$, where t_{base}^{BT} denotes the parameter t_{base} for BT .

Step 2: For a node x in $\text{rep}(\theta_1 - t_{base}^{BT}, \theta_2 - t_{base}^{BT})$, $base(x) := base(x) + c$, $ceil(x) := ceil(x) + c$, and $h_e(x) := h_e(x) + c$.

Step 3: For a node x such that $t^l(x) = \theta_1 - t_{base}^{BT}$, call $\text{NORMALIZE}(BT, x)$.

If $base(x^-) = base(x)$ (i.e., $\hat{h}(x^-) = \hat{h}(x)$), then $y := x^-$,

$$\begin{aligned} base(y) &:= \frac{1}{t^r(y^+) - t^l(y)} \left(h(y)(t^r(y) - t^l(y)) \right. \\ &\quad \left. + h(y^+)(t^r(y^+) - t^l(y^+)) \right) \\ t^r(y) &:= t^r(y^+) \\ t^l(y^+) &= t^r(y^+) \text{ (i.e., } y^+ \text{ becomes dummy).} \end{aligned} \quad (11)$$

and call $\text{NORMALIZE}(BT, y)$ and $\text{NORMALIZE}(BT, y^+)$.

Step 4: For a leaf y such that $t^r(y) = \theta_2 - t_{base}^{BT}$, call $\text{NORMALIZE}(BT, y)$.

If $base(y) = base(y^+)$ (i.e., $\hat{h}(y) = \hat{h}(y^+)$), then update $base(y)$, $t^r(y)$, $t^l(y^+)$ and $t^r(y^+)$ as (11), and call $\text{NORMALIZE}(BT, y)$ and $\text{NORMALIZE}(BT, y^+)$. \square

Steps 3 and 4 are performed to keep (6). Note that $h_e(x)$ is updated in Step 2 for all nodes in $\text{rep}(\theta_1 - t_{base}^{BT}, \theta_2 - t_{base}^{BT})$. It follows from (9) that $h_e(y)$ must be updated for all proper ancestors y of a node in $\text{rep}(\theta_1 - t_{base}^{BT}, \theta_2 - t_{base}^{BT})$. Since a proper ancestor y of some node in $\text{rep}(\theta_1 - t_{base}^{BT}, \theta_2 - t_{base}^{BT})$ is a proper ancestor of the node x such that $t^l(x) = \theta_1 - t_{base}^{BT}$ or $t^r(x) = \theta_2 - t_{base}^{BT}$, all such $h_e(y)$'s are updated in Steps 3 and 4 by operation NORMALIZE .

Operation $\text{SPLIT}(BT, t : \text{a nonnegative real})$

Step 1: Find a node x such that $t^l(x) \leq t < t^r(x)$.

Step 2: Call $\text{NORMALIZE}(BT, x^-)$ and $\text{NORMALIZE}(BT, x)$.

Step 3: If $t^l(x) = t$, then halt.

Step 4: For the node $y \in \{x^-, x\}$ such that $t^l(y) \leq t < t^r(y)$, construct the left child y_1 with $t^l(y_1) := t^l(y)$, $t^r(y_1) := t$, $base(y_1) := 0$ and $ceil(y_1) := +\infty$, and construct the right child y_2 with $t^l(y_2) := t$, $t^r(y_2) := t^r(y)$, $base(y_2) := 0$ and $ceil(y_2) := +\infty$.

Step 5: Call $\text{NORMALIZE}(BT, y_1)$ and $\text{NORMALIZE}(BT, y_2)$. \square

We can see that the following two lemmas hold.

Lemma 4.3: *Let BT be a valid binary balanced tree representing a table g , and let t be a nonnegative real. Then BT' obtained by operation $\text{SPLIT}(BT, t)$ is a valid binary balanced tree representing g . \square*

Lemma 4.4: *Let BT be a valid binary balanced tree representing a table g , and let $I = ([\theta_1, \theta_2], c)$ be a time interval. Then $\text{ADD}(BT, \theta_1, \theta_2, c)$ produces a valid binary balanced tree representing the table $g + I$, and moreover, it can be handled in $O(\text{height}(BT))$ time. \square*

4.4. Operation Ceil-Table

This section considers operation *Ceil-Table*. Let BT be a valid binary balanced tree representing a table g and let c be an upper bound of BT . As mentioned in Section 4.1, we set $ceil(r^{BT}) = c$, and modify BT so that $\hat{h}(x) \neq \hat{h}(x^+)$ holds for any two consecutive active leaves x and x^+ .

Operation $\text{CEIL}(BT, c : \text{a positive real})$

Step 1: Compute the leftmost active leaf y such that $h(y) - e(y) \geq c$ by using h_e . If BT has no such node, then go to Step 5.

Step 2: Call $\text{NORMALIZE}(BT, y)$, and update $\text{base}(y)$, $t^r(y)$, $t^l(y^+)$ and $t^r(y^+)$ as (11).

Step 3: Call $\text{NORMALIZE}(BT, y)$ and $\text{NORMALIZE}(BT, y^+)$. Return to Step 1.

Step 4: For a root r^{BT} , $\text{ceil}(r^{BT}) := c$. \square

Lemma 4.5: *Let BT be a valid binary balanced tree representing a table g , and let c be a nonnegative real. Then BT' obtained by operation $\text{CEIL}(BT, c)$ is a valid binary balanced tree representing the table obtained from g by ceiling it by c .* \square

Step 3 concatenates two consecutive active leaves x and x^+ , where x^+ becomes dummy. We notice that the active leaf x (which has already been concatenated) may further be concatenated. This means that $\hat{h}(x) = \hat{h}(x^+)$ may hold after successive concatenations, even if original BT satisfies $\hat{h}(x) \neq \hat{h}(x^+)$.

5. Time complexity of SINGLE-PHASE with our data structures

In this section, we analyze the complexity of Algorithm SINGLE-PHASE with our data structures. Recall that the algorithm only applies to tables A_v and/or S_v the following three basic operations: *Add-Table* (i.e., adding tables), *Shift-Table* (i.e., shifting a table), and *Ceil-Table* (i.e., ceiling a table by a prescribed capacity c).

Lemma 5.1: *All Shift-Table's in SINGLE-PHASE require $O(n)$ time in total.* \square

Lemma 5.2: *All Add-Table's in SINGLE-PHASE require $O(n \log^2 n)$ time in total.* \square

Lemma 5.3: *All Ceil-Table's in SINGLE-PHASE require $O(n \log n)$ time in total.* \square

From lemmas above, we have the following result.

Theorem 5.4: *The sink location problem on dynamic tree networks can be solved in $O(n \log^2 n)$ time.* \square

If a given network is a tree and has a single sink, we can show the following corollary.

Corollary 5.5: *If a given network is tree and has a single sink, SINGLE-PHASE can solve the quickest transshipment problem in $O(n \log^2 n)$ time.* \square

6. Conclusions

In this paper, we have developed an $O(n \log^2 n)$ time algorithm for a sink location problem for dynamic flows in a tree network. This improves upon an $O(n^2)$ time algorithm in [12].

We have considered continuous-time dynamic flows that allow intermediate storage at vertices. We note that

optimal sinks remain the same, even if we do not allow intermediate storage, and moreover, our algorithm can also be applicable for discrete-time dynamic flows. Therefore, our sink location problem is solvable in $O(n \log^2 n)$ time for dynamic continuous-time/discrete-time flows with/without intermediate storage.

Acknowledgements

This research is partially supported by the Grant-in-Aid for Creative Scientific Research of the Ministry of Education, Culture, Sports, Science and Technology.

References

- [1] K. Arata, S. Iwata, K. Makino and S. Fujishige: Locating sources to meet flow demands in undirected networks, *Journal of Algorithms*, **42** (2002) 54–68.
- [2] J. E. Aronson: A survey of dynamic network flows, *Annals of Operations Research*, **20** (1989) 1–66.
- [3] L. G. Chalmet, R. L. Francis and P. B. Saunders: Network models for building evacuation. *Management Science*, **28** (1982) 86–105.
- [4] L. Fleischer and É. Tardos: Efficient continuous-time dynamic network flow algorithms, *Operations Research Letters*, **23** (1998) 71–80.
- [5] L. R. Ford, Jr. and D. R. Fulkerson: Constructing maximal dynamic flows from static flows, *Op. Res.*, **6** (1958) 419–433.
- [6] L. R. Ford, Jr. and D. R. Fulkerson: *Flows in Networks*, (Princeton University Press, Princeton, NJ, 1962).
- [7] H. W. Hamacher and S.A.Tjandra: Mathematical modelling of evacuation problems: A state of the art. In: *Pedestrian and Evacuation Dynamics*, Springer, (2002) 227–266.
- [8] B. Hoppe and É. Tardos: Polynomial Time Algorithms for some evacuation problems, In: *Proc. of 5th Ann. ACM-SIAM Symp. on Discrete Algorithms*, (1994) 433–441.
- [9] B. Hoppe and É. Tardos: The quickest transshipment problem, *Mathematics of Operations Research*, **25** (2000) 36–62.
- [10] S. Iwata, L. Fleischer, and S. Fujishige: A combinatorial strongly polynomial algorithm for minimizing submodular functions, *Journal of the ACM*, **48** (2001) 761–777.
- [11] H. Ito, H. Uehara and M. Yokoyama: A faster and flexible algorithm for a location problem on undirected flow networks, *IEICE Trans. Fundamentals*, **E83-A** (2000) 704–712.
- [12] S. Mamada, K. Makino and S. Fujishige: Optimal sink location problem for dynamic flows in a tree network, *IEICE Trans. Fundamentals*, **E85-A** (2002) 1020–1025.
- [13] P. B. Mirchandani and R. L. Francis: *Discrete Location Theory* (John Wiley & Sons, Inc., 1989).
- [14] W. B. Powell, P. Jaillet, and A. Odoni: Stochastic and dynamic networks and routing, In: *Network Routing, Handbooks in Operations Research and Management Science* **8** (M. O. Ball, et al., eds, North-Holland, Amsterdam, The Netherlands, 1995), Chapter 3, 141–295.
- [15] A. Schrijver: A combinatorial algorithm minimizing submodular functions in strongly polynomial time, *J. Combinatorial Theory*, **B80** (2000) 346–355.
- [16] H. Tamura, M. Sengoku, S. Shinoda, and T. Abe: Some covering problems in location theory on flow networks, *IEICE Trans. Fundamentals*, **E75-A** (1992) 678–683.
- [17] H. Tamura, H. Sugawara, M. Sengoku, and S. Shinoda: Plural cover problem on undirected flow networks, *IEICE Trans. Fundamentals*, **J81-A** (1998) 863–869 (in Japanese).