AN $\mathrm{O}(n \log^2 n)$ ALGORITHM FOR A SINK LOCATION PROBLEM IN DYNAMIC TREE NETWORKS

Satoko Mamada,[1] Takeaki Uno,[2] Kazuhisa Makino,[1] and Satoru Fujishige[3]

[1] *Division of Mathematical Science for Social Systems, Graduate School of Engineering Science, Osaka University*
mamada@inulab.sys.es.osaka-u.ac.jp, makino@sys.es.osaka-u.ac.jp

[2] *Foundations of Informatics Research Division, National Institute of Informatics*
uno@nii.jp

[3] *Research Institute for Mathematical Sciences, Kyoto University*
fujishig@kurims.kyoto-u.ac.jp

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 quickly as possible. We present an $\mathrm{O}(n \log^2 n)$ time algorithm for the sink location problem in a dynamic network of tree structure, where n is the number of vertices in the network. This improves upon the existing $\mathrm{O}(n^2)$-time bound. As a corollary, we also show that the quickest transshipment problem can be solved in $\mathrm{O}(n \log^2 n)$ time if a given network is a tree and has a single sink. Our results are based on data structures for representing tables (i.e., sets of intervals with their height), which may be of independent interest.

Keywords: Dynamic flows, location problem, tree networks.

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 dynamic 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 the 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 who 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 this 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, 10, 16].

We consider the sink location problem in dynamic *tree* networks. This is because some production systems and underground passages form almost-tree networks. Moreover, one of the ideal evacuation plans makes everyone to be evacuated fairly and without confusion. For such a purpose, it is natural to assume that the possible evacuation routes form a tree. We finally mention that the multi-sink location problem can be solved by solving the (single-)sink location problem polynomially many times [12]. It is known [11] that the problem can be solved in $\mathrm{O}(n^2)$ time by using a double-phase algorithm, where n denotes the number of vertices in the given network. We show that the problem is solvable in $\mathrm{O}(n \log^2 n)$ time.

Our algorithm is based on a simple single-phase procedure, but uses sophisticated data structures for representing 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 $\mathrm{O}((\sum_i d_i) \log^2(\sum_i d_i))$ 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 $\mathrm{O}(n \log^2 n)$ 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 [8]. However, since their algorithm makes use of submodular function minimization [9, 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 $\mathrm{O}(n \log^2 n)$ time if the 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 discusses our data structures and shows the complexity of our single-phase algorithm with our data structures. Finally, Section 5 gives some conclusions.

Due to the space limitations, some proofs have been omitted.

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 a tree T, where $c : E \to \mathbf{R}_+$ is a capacity function representing the least upper bound for the rate of flow through each edge per unit time, $\tau : E \to \mathbf{R}_+$ a transit time function, and $b : V \to \mathbf{R}_+$ 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 quickly 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 $\theta \in \mathbf{R}_+$,

$$0 \le f_e(\theta) \le c(e). \tag{1}$$

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

$$\sum_{e\in\delta^+(v)}\int_0^{\Theta} f_e(\theta)d\theta - \sum_{e\in\delta^-(v)}\int_{\tau(e)}^{\Theta} f_e(\theta-\tau(e))d\theta \le b(v). \qquad (2)$$

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

$$\sum_{e\in\delta^-(v^*)}\int_{\tau(e)}^{\Theta} f_e(\theta-\tau(e))d\theta - \sum_{e\in\delta^+(v^*)}\int_0^{\Theta} f_e(\theta)d\theta = \sum_{v\in V\setminus\{v^*\}} b(v). \qquad (3)$$

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 (3), 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, 8, 11], for example).

3. A Single-Phase Algorithm

This section presents a simple $\mathrm{O}(n^2)$ algorithm with a single phase. Because of the simplicity, it gives us a good basis for developing a faster algorithm. In fact, we can construct an $\tilde{\mathrm{O}}(n)$ algorithm based on this framework, which is given in the next section.

The algorithm computes two tables, *Arriving Table* A_v and *Sending Table* S_v for each vertex $v \in V$. Let us assume that a sink t is given for a while, in order to explain them. Arriving Table A_v represents the sum of the flow rates arriving at vertex v as a function of time θ, i.e.,

$$\sum_{e\in\vec{E}(t):e=(u,v)} f_e(\theta-\tau(e)) + \eta_\theta(v), \qquad (4)$$

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 $0 \le \theta < \Delta$; otherwise 0. Here, Δ denotes a sufficiently small positive constant. Intuitively, $\eta_\theta(v)$ denotes the initial supply at v. Sending Table S_v represents the flow rate leaving vertex v as a function of time θ, i.e.,

$$f_{(v,w)}(\theta), \qquad (5)$$

where $(v,w) \in \vec{E}(t)$.

Let us consider a table $g : \mathbf{R}_+ \to \mathbf{R}_+$, 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 toward an optimal sink (which will be computed), we only consider the table g which is representable as

$$g(\theta) = \begin{cases} 0 & \text{if } \theta < \theta_1 \\ g(\theta_i) & \text{if } \theta_i \leq \theta < \theta_{i+1} \quad \text{for } i = 1, \cdots, k-1 \\ 0 & \text{if } \theta \geq \theta_k, \end{cases} \tag{6}$$

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

$$((-\infty, \theta_1), 0), \quad ([\theta_i, \theta_{i+1}), g(\theta_i)) \;\; (i = 1, 2, \cdots, k), \tag{7}$$

where $\theta_{k+1} = +\infty$ and $g(\theta_k) = 0$.

Intuitively, our single-phase algorithm first constructs Sending Table S_v for each leaf v to send $b(v)$ to its adjacent vertex. Then the algorithm removes a leaf v^* from T such that the completion time of S_v is the smallest, since T has an optimal sink other than v^*. If some vertex v becomes a leaf of the resulting tree T, then the algorithm computes Sending Table S_v to send all the supplies that have already arrived at v to an adjacent vertex $p(v)$ of the resulting tree T, by using Sending Tables for the vertices w ($\neq p(v)$) that are adjacent to v in the original tree. 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

Input: A tree network $\mathcal{N} = (T = (V, E), c, \tau, b)$.

Output: An optimal sink t that has the minimum completion time $C(t)$ among all vertices of T.

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 $v^* \in L$ minimizing *Time*$(v, p(v))$, i.e., *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_w for the vertices w 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. □

Here $T[W]$ denotes a subtree of T induced by a vertex set W. Note that $p(v)$'s in Steps 1 and 2 are uniquely defined, since v's are leaves of $T[W]$.

We then have the following lemma, though we skip the proof.

LEMMA 1 *Algorithm* SINGLE-PHASE *outputs an optimal sink* t. □

If we construct Arriving and Sending Tables explicitly, each table g can be computed in time linear in the total number of intervals in the tables from which g is constructed. Since the number of intervals in each table is linear in n,[1] Algorithm SINGLE-PHASE requires $\mathrm{O}(n^2)$ time. In Section 4, we present a method to represent these tables implicitly, and develop an $\mathrm{O}(n \log^2 n)$ time algorithm for our location problem.

4. Implicit Representation for Arriving and Sending Tables

Since Algorithm SINGLE-PHASE requires $\Theta(n^2)$ time if explicit representations are used for tables, we need 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). However, standard interval trees cannot efficiently handle operation *Ceil-Table*. This paper develops new interval trees which efficiently handle all the three operations.

Data Structures for Implicit Representation

This section explains our data structures for representing tables which are obtained from interval trees by attaching several parameters to handle the three operations efficiently. Let g be a table represented as

$$I_i = ([\theta_i, \theta_{i+1}), g(\theta_i)) \quad (i = 0, 1, \cdots, k), \tag{8}$$

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 MAKE-TREE 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 (8), 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 internal node x, $t^l(x)= \min_{y \in Leaf(x)} t^l(y)$, $t^r(x)= \max_{y \in Leaf(x)} t^r(y)$, $base(x) = 0$ 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 internal node 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. □

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 $\mathrm{ADD}(BT_1; \theta_1, \theta_2, c)$, 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. Then we add an interval $([\theta_1, \theta_2), c)$ to the resulting $\widetilde{BT}_{g_1}$. One of the simplest way is to add c to all leaves of $\widetilde{BT}_{g_1}$ such that the corresponding intervals are included in $[\theta_1, \theta_2)$. However, this takes $\mathrm{O}(n)$ time, since BT_{g_1} may have $\mathrm{O}(n)$ such intervals. We therefore add c only to their representatives.

Note 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 thus update $base$ of $\widetilde{BT}_{g_1}$ as follows

$$base(x) := base(x) + c \quad \text{for all } x \in rep(\theta_1, \theta_2). \tag{9}$$

We remark that this is a standard technique for interval tree. 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 $y_1(= x)$, $y_2, \cdots$, $y_s(= r^{BT})$ 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

$$h(x) = \sum_{i=1}^{s} base(y_i). \tag{10}$$

Operation $\mathrm{ADD}(BT_{g_1}; \theta_1, \theta_2, c)$ can be handled in O(*height*(BT_{g_1})) time, since $|rep\ (\theta_1, \theta_2)| \leq 2height\ (BT_{g_1})$. This means that BT_g can be constructed from BT_{g_1} and BT_{g_2} in O $(\#(BT_{g_2}) \log n)$ time by balancing the tree after each addition. 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 for the details). Thus *Add-Table* can be performed efficiently.

However, operations *Ceil-Table* in Algorithm SINGLE-PHASE require Θ (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 (10) does not represent the actual height any longer. Roughly speaking, the actual height is c if $c \leq h(x)$, and $h(x)$, otherwise. We call $h(x)$ the *tentative height* of x in BT, and denote by $\hat{h}(x)$ the *actual height* of x. Let us consider a scenario that an interval $([\theta_1, \theta_2), c')$ is added to BT after bounding it by c. Let x be an active leaf such that (i) the corresponding interval is contained in $[\theta_1, \theta_2)$ and (ii) the actual height is c, immediately after bounding BT by c. Then we note that the actual height of x is $c + c'$ after the scenario, which is different from both $h(x)$ and c. To deal with such scenarios, we update $ceil$ to compute the actual height $\hat{h}(x)$ efficiently (See more details in the subsequent sections). The actual height $\hat{h}(x)$ can be computed as

$$\hat{h}(x) = h(x) - \max_{y \in path(x, r^{BT})} \{0, \Big(\sum_{z \in path(x,y)} base(z) \Big) - ceil(y)\}, \tag{11}$$

where $path(x, y)$ denotes the path from x to y. 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 $\textsc{MakeTree}$.

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

$$\hat{h}(x) \neq \hat{h}(x'). \tag{12}$$

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.,

$$t^r(x^-) = t^l(x), \quad t^l(x^+) = t^r(x). \tag{13}$$

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

$$\hat{t}^r(x) = t_{base} + t^r(x) + (t^r(x) - t^l(x)) \times \frac{h(x) - \hat{h}(x)}{\hat{h}(x) - \hat{h}(x^+)} \tag{14}$$

$$\hat{t}^l(x) = \hat{t}^r(x^-). \tag{15}$$

Here $\hat{t}^r(x)$ and $\hat{t}^l(x)$ are well-defined from 12. 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 (11), (14), and (15) 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 $\mathrm{O}(height(BT))$ time. In order to check (12) efficiently, each active leaf x has

$$e(x) = \begin{cases} \max\{0, h(x) - h(x^+)\} \times \dfrac{t^r(x^+) - t^r(x)}{t^r(x^+) - t^l(x)} & \text{if } x^+ \text{ exists,} \\ +\infty & \text{otherwise,} \end{cases} \tag{16}$$

and each node x has

$$h_e(x) = \max_{y \in Leaf_A(x)} \{ \Big(\sum_{z \in path(x,y)} base(z) \Big) - e(y) \}, \tag{17}$$

where $Leaf_A(x)$ denotes the set of active leaves that are descendants of x, and $path\ (x, y)$ denotes the set of nodes on the path from x to y. Thus we have the following lemma.

LEMMA 2 *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 $h(x) - e(x) < c$, and*

(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 $\mathrm{O}(height\,(BT))$ 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.

$$h_e(x) = \begin{cases} base(x) - e(x) & \text{if } x \text{ is a leaf,} \\ \max\{h_e(x_1), h_e(x_2)\} + base(x) & \text{otherwise,} \end{cases} \tag{18}$$

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 internal node x satisfies $t^l(x) = \min_{y \in Leaf(x)} t^l(y)$, and $t^r(x) = \max_{y \in Leaf(x)} t^r(y)$.

(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.

Operation NORMALIZE

As discussed in Section 4, 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

$$base(y) := \begin{cases} 0 & \text{for a proper ancestor } y \text{ of } x^- \text{ or } x \\ \hat{h}(y) & \text{for } y = x^- \text{ or } x \end{cases} \tag{19}$$

$$ceil(y) := +\infty \quad \text{for an ancestor } y \text{ of } x^- \text{ or } x \tag{20}$$

$$t^r(y) = t^l(y^+) := \hat{t}^r(y) \quad \text{for } y = x^- \text{ or } x.$$

In fact, we perform this operation, when we insert a leaf x or change the parameters $ceil(x)$, $base(x)$, $t^r(x)$, and $t^l(x)$ of a leaf x. The following operation, called NORMALIZE, updates BT as above, and also maintains the balance of BT (i.e., *height*$(BT) = \mathrm{O}(\log n)$).

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,

$$base(z_i) := base(z_i) + base(z),\ \ ceil(z_i) := \min\{ceil(z_i) + base(z), ceil(z)\},$$
$$base(z) := 0,\ \ ceil(z) := +\infty.$$

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 $t^r(y) = t^l(y^+) := \hat{t}^r(y)$ and $base(y) := ceil(y)$. 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}. □

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

LEMMA 3 *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* NORMALIZE(BT, x) *is a valid binary balanced tree that represents g and satisfies* (20). *Furthermore, BT' is computable from BT in $\mathrm{O}(height(BT))$ time.* □

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 $\hat{h}(x)$. □

Operation ADD$(BT, \theta_1, \theta_2, c)$

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

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

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

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

$$\begin{aligned} y &:= x^-, \\ t^r(y) = t^l(y^+) &:= t^r(y^+) \text{ (i.e., } y^+ \text{ becomes dummy).} \end{aligned} \tag{21}$$

and call $\textsc{Normalize}(BT, y)$ and $\textsc{Normalize}(BT, y^+)$.

Step 4: For a leaf y such that $t^r(y) = \theta_2 - t^{BT}_{base}$, call $\textsc{Normalize}(BT, y)$.
If $base(y) = base(y^+)$ (i.e., $\hat{h}(y) = \hat{h}(y^+)$), then update $t^r(y)$, $t^l(y^+)$ and $t^r(y^+)$ as 21, and call $\textsc{Normalize}(BT, y)$ and $\textsc{Normalize}(BT, y^+)$. □

Steps 3 and 4 are performed to keep (12). Note that $h_e(x)$ is updated in Step 2 for all nodes in $rep(\theta_1 - t^{BT}_{base}, \theta_2 - t^{BT}_{base})$. It follows from (18) that $h_e(y)$ must be updated for all proper ancestors y of a node in $rep(\theta_1 - t^{BT}_{base}, \theta_2 - t^{BT}_{base})$. Since a proper ancestor y of some node in $rep(\theta_1 - t^{BT}_{base}, \theta_2 - t^{BT}_{base})$ is a proper ancestor of the node x such that $t^l(x) = \theta_1 - t^{BT}_{base}$ or $t^r(x) = \theta_2 - t^{BT}_{base}$, all such $h_e(y)$'s are updated in Steps 3 and 4 by operation $\textsc{Normalize}$.

Operation $\textsc{Split}(BT, t$: *a nonnegative real*$)$

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

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

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 $\textsc{Normalize}(BT, y_1)$ and $\textsc{Normalize}(BT, y_2)$. □

We can see that the following two lemmas hold.

LEMMA 4 *Let BT be a valid binary balanced tree representing a table g, and let t be a nonnegative real. Then BT' obtained by operation $\textsc{Split}(BT, t)$ is a valid binary balanced tree representing g in $\mathrm{O}(height(BT))$ time.* □

LEMMA 5 *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 $\textsc{Add}(BT, \theta_1, \theta_2, c)$ produces a valid binary balanced tree representing the table $g + I$, and moreover, it can be handled in $\mathrm{O}(height(BT))$ time.* □

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, 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 $\textsc{Ceil}(BT, c$: *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 4.

Step 2: Call $\textsc{Normalize}(BT, y)$ and $\textsc{Normalize}(BT, y^+)$,

$$base(y) := \frac{base(y)(t^r(y) - t^l(y)) + base(y^+)(t^r(y^+) - t^l(y^+))}{t^r(y^+) - t^l(y)}, \text{ and}$$

$t^r(y) = t^l(y^+) := t^r(y^+)$.

Step 3: Call $\text{NORMALIZE}(BT, y)$ and $\text{NORMALIZE}(BT, y^+)$. Return to Step1.

Step 4: For a root r^{BT}, *ceil*$(r^{BT}) := c$. □

LEMMA 6 *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.* □

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 $\hat{h}(x) \neq \hat{h}(x^+)$.

Time complexity of SINGLE-PHASE with our data structures

We can see that all operations *Add-Tables*, *Shift-Tables*, and *Ceil-Tables* can be done in $\mathrm{O}(n \log^2 n)$ time in total, though we skip its proof.

THEOREM 7 *The sink location problem in dynamic tree networks can be solved in $\mathrm{O}(n \log^2 n)$ time.* □

This implies the following corollary.

COROLLARY 8 *If a given network is tree and has a single sink,* SINGLE-PHASE *can solve the quickest transshipment problem in $\mathrm{O}(n \log^2 n)$ time.* □

5. Conclusions

In this paper, we have developed an $\mathrm{O}(n \log^2 n)$ time algorithm for a sink location problem for dynamic flows in a tree network. This improves upon an $\mathrm{O}(n^2)$ time algorithm in [11].

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 $\mathrm{O}(n \log^2 n)$ time for dynamic continuous-time/discrete-time flows with/without intermediate storage.

Acknowledgments

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

Notes

1. It was shown in [11] that the number of intervals is at most $3n$ for *discrete-time* dynamic flows.
2. For simplicity, we write the first interval I_0 as $([-\infty, \theta_1), 0)$ instead of $((-\infty, \theta_1), 0)$.

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 OR*, **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: The quickest transshipment problem, *Mathematics of Operations Research*, **25** (2000) 36–62.

[9] S. Iwata, L. Fleischer, and S. Fujishige: A combinatorial strongly polynomial algorithm for minimizing submodular functions, *Journal of the ACM*, **48** (2001) 761–777.

[10] 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.

[11] 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.

[12] S. Mamada, T, Uno, K. Makino, and S. Fujishige: An evacuation problem in tree dynamic networks with multiple exits, Working paper.

[13] P. B. Mirchandani and R. L. Francis: *Discrete Location Theory*, (John Wile & 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, T. L. Magnanti, C. L. Monma, and G. L. Nemhauser, 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.