RA-004

An LP-Based Heuristic Algorithm for the Node Capacitated In-tree Packing Problem

田中 勇真† 佐々木 美裕‡ 柳浦 睦憲†
Yuma Tanaka Mihiro Sasaki Mutsunori Yagiura

1 Introduction

In this paper, we consider *the node capacitated in-tree packing problem*. The input consists of a directed graph, a root node, a node capacity function and edge consumption functions for heads and tails. The problem consists of finding the maximum number of rooted in-trees such that the total consumption of the in-trees at each node does not exceed the capacity of the node.

Let $G = (V, E)$ be a directed graph, $r \in V$ be a root node and $\mathbb{R}_+$ be the set of nonnegative real numbers. In addition, let $t : E \to \mathbb{R}_+$ and $h : E \to \mathbb{R}_+$ be tail and head consumption functions on directed edges, respectively, and $b_i \in \mathbb{R}_+$ be the capacity of a node $i \in V$. For convenience, we define $T_{\rm all}$ as the set of all in-trees rooted at the given root $r \in V$ in the graph G. Let $\delta_j^+(i)$ (resp., $\delta_j^-(i)$) be the set of edges in an in-tree $j \in T_{\rm all}$ leaving (resp., entering) a node $i \in V$. The consumption a_{ij} of an in-tree $j \in T_{\rm all}$ at a node $i \in V$ is defined as

$$a_{ij} = \sum_{e \in \delta_j^+(i)} t(e) + \sum_{e \in \delta_j^-(i)} h(e). \qquad (1)$$

We call the first term of the above equation (1) *tail consumption*, and the second term *head consumption*. The node capacitated in-tree packing problem is to find a subset $T \subseteq T_{\rm all}$ of in-trees and the packing number x_j of each in-tree $j \in T$ subject to the node capacity restriction

$$\sum_{j \in T} a_{ij} x_j \le b_i, \quad \forall i \in V, \qquad (2)$$

so as to maximize the total number of packed in-trees $\sum_{j \in T} x_j$.

This problem is known to be NP-hard (Imahori et al., 2009). Furthermore, it is still NP-hard even if instances are restricted to complete graphs embedded in a space with tail consumptions depending only on the distance between end nodes.

This problem is studied in the context of sensor networks. Recently, several kinds of graph packing problems are studied in the context of ad hoc wireless networks and sensor networks. These problems are called *network lifetime problems*. The important problems included among this category are the node capacitated spanning subgraph packing problems (Calinescu et al., 2003; Heinzelman et al., 2002; Sasaki et al., 2007). For sensor networks, for example, a spanning subgraph corresponds to a communication network topology for collecting information from all nodes (sensors) to the root (base station) or for sending information from the root to all other nodes. Sending a message along an edge consumes energy at end nodes, usually depending on the distance between them. The use of energy for each sensor is severely limited because the sensors use batteries. It is therefore important to design the topologies for communication in order to save energy consumption and make sensors operate as long as possible. For this problem, Heinzelman et al. (2002) proposed an algorithm, called LEACH-C (low energy adaptive clustering hierarchy centralized), that uses arborescences with limited height for communication topologies. For more energy effcient communication networks, a multiround topology construction problem was formulated as an integer programming problem, and a heuristic solution method was proposed by Sasaki et al. (2007). In the formulation of Calinescu et al. (2003), head consumptions are not considered, and the consumption at each node is the maximum tail consumption among the edges leaving the node. There are variations of the problem with respect to additional conditions on the spanning subgraph such as strong connectivity, symmetric connectivity, and directed out-tree rooted at a given node. Calinescu et al. (2003) discussed the hardness of the problem and proposed several approximation algorithms.

These network lifetime problems are similar to the well-known edge-disjoint spanning arborescence packing problem: Given a directed graph $G = (V, E)$ and a root $r \in V$, find the maximum number of edge-disjoint spanning arborescences rooted at r. The edge-disjoint spanning arborescence packing problem is solvable in polynomial time (Edmonds, 1973; Gabow and Manu, 1998). Its capacitated version is also solvable in polynomial time (Korte and Vygen, 2007; Mader, 1981; Schrijver, 2003).

In this paper, we propose a two-phase heuristic algorithm for the node capacitated in-tree packing problem. In the first phase, it generates candidate in-trees to be packed. The node capacitated in-tree packing problem can be formulated as an IP (integer programming) problem, and the proposed algorithm employs the delayed column generation method for the LP-relaxation of the problem to generate promising can-

†Graduate School of Information Science, Nagoya University
‡Faculty of Information Sciences and Engineering, Nanzan University

didate in-trees. In the second phase, the algorithm computes the packing number of each in-tree. Our algorithm solves this second-phase problem by first modifying feasible solutions of the LP-relaxation problem and then improving them with a greedy algorithm. We conducted computational experiments on benchmark instances and on randomly generated instances with up to 200 nodes. The results show that the proposed algorithm obtains solutions that deviate at most 0.78% from upper bounds, and comparisons with another approach from the literature show that our method works more effectively for this problem.

2 Formulation

A node capacitated in-tree packing problem can be formulated as the following IP problem:

$$\begin{aligned} &\text{maximize} && \sum_{j\in T_{\rm all}} x_j, \\ &\text{subject to} && \sum_{j\in T_{\rm all}} a_{ij}x_j \le b_i, \quad \forall i \in V, \\ & && x_j \ge 0, \qquad x_j \in \mathbb{Z},\ \forall j \in T_{\rm all}. \end{aligned} \tag{3}$$

where the notations are summarized as follows:

- V: the set of nodes,
- $T_{\rm all}$: the set of all in-trees rooted at the given root $r \in V$,
- a_{ij}: the consumption (defined by equation (1)) of an arborescence $j \in T_{\rm all}$ at a node $i \in V$,
- b_i: the capacity of a node $i \in V$,
- x_j: the packing number of an in-tree $j \in T$,
- $\mathbb{Z}$: the set of all integers.

We define $T_{\rm all}$ as the set of all in-trees rooted at the given root $r \in V$. However, the number of in-trees in $T_{\rm all}$ can be exponentially large, and it is difficult in practice to handle all of them. We therefore consider a subset $T \subseteq T_{\rm all}$ of in-trees and deal with the following problem:

$$\begin{aligned} P(T)\quad &\text{maximize} && \sum_{j\in T} x_j, \\ &\text{subject to} && \sum_{j\in T} a_{ij}x_j \le b_i, \quad \forall i \in V, \\ & && x_j \ge 0, \qquad x_j \in \mathbb{Z},\ \forall j \in T. \end{aligned}$$

If $T = T_{\rm all}$, the problem $P(T_{\rm all})$ is equivalent to the original problem (3). We denote the optimal value of $P(T)$ by $\mathrm{OPT}_{P(T)}$.

We also consider the LP relaxation problem of $P(T)$, which is formally described as follows:

$$\begin{aligned} LP(T)\quad &\text{maximize} && \sum_{j\in T} x_j, \\ &\text{subject to} && \sum_{j\in T} a_{ij}x_j \le b_i, \quad \forall i \in V, \\ & && x_j \ge 0, \qquad \forall j \in T. \end{aligned}$$

When $T = T_{\rm all}$, the problem $LP(T_{\rm all})$ is the LP relaxation of the original problem (3). We denote the optimal value of $LP(T)$ by $\mathrm{OPT}_{LP(T)}$.

In general, $\lfloor \mathrm{OPT}_{LP(T)} \rfloor$ (where $\lfloor x \rfloor$ stands for the floor function of x) gives an upper bound of $\mathrm{OPT}_{P(T)}$ because $\mathrm{OPT}_{P(T)}$ is an integer. Note that if $T \neq T_{\rm all}$, then $\mathrm{OPT}_{LP(T)}$ is not necessarily an upper bound of $\mathrm{OPT}_{P(T_{\rm all})}$. For convenience, denote the vector of variables x_j for all $j \in T$ by $(x_j \mid j \in T)$. Then, for any feasible solution $(x_j \mid j \in T)$ of $LP(T)$, $(\lfloor x_j \rfloor \mid j \in T)$ is a feasible solution of $P(T_{\rm all})$ and its objective value is a lower bound of $\mathrm{OPT}_{P(T_{\rm all})}$.

3 In-trees Generating Algorithm

3.1 Pricing problem

We employ the delayed column generation method to generate candidate in-trees to be packed. It starts from an arbitrary set $T \subseteq T_{\rm all}$, and repeatedly augments the set T until some stopping criterion is satisfied. To apply the delayed column generation method, we consider the following dual of the LP relaxation problem $LP(T)$:

$$\begin{aligned} D(T)\quad &\text{minimize} && \sum_{i\in V} b_i\lambda_i, \\ &\text{subject to} && \sum_{i\in V} a_{ij}\lambda_i \ge 1, \quad \forall j \in T, \\ & && \lambda_i \ge 0, \qquad \forall i \in V. \end{aligned}$$

When $T = T_{\rm all}$, the problem $D(T_{\rm all})$ is the dual of the LP relaxation problem $LP(T_{\rm all})$. Thus, the pricing problem of $LP(T)$ is defined as the problem of finding an in-tree $\tau \in T_{\rm all} \setminus T$ that satisfies

$$\sum_{i\in V} a_{i\tau}\lambda^*_i < 1, \tag{4}$$

where $(\lambda^*_i \mid i \in V)$ is an optimal dual solution of the problem with the current T. If there is no in-tree which satisfies condition (4), then the optimal value of the problem $LP(T)$ cannot be improved any more. On the other hand, if condition (4) is satisfied by a certain in-tree $\tau \in T_{\rm all} \setminus T$, then the optimal value of the problem $LP(T)$ can be improved by adding the in-tree τ into T.

Let E_j be the set of all edges in each in-tree $j \in T_{\rm all}$, and $\phi(vw) := \lambda^*_v t(vw) + \lambda^*_w h(vw)$ be the cost of each edge $vw \in E$. Defining $a_{i\tau}$ as in equation (1), it is possible to transform the left-hand side of (4) as follows:

$$\begin{aligned} \sum_{i\in V} a_{i\tau}\lambda^*_i &= \sum_{i\in V} \lambda^*_i \left\{ \sum_{e\in\delta^+_\tau(i)} t(e) + \sum_{e\in\delta^-_\tau(i)} h(e) \right\} \\ &= \sum_{vw\in E_\tau} \{\lambda^*_v t(vw) + \lambda^*_w h(vw)\} \\ &= \sum_{vw\in E_\tau} \phi(vw). \end{aligned} \tag{5}$$

Thereby, after calculating the minimum value of equation (5), we know if there is a possibility of improving the optimal value of problem $LP(T)$ by adding an in-tree τ. Given the costs of edges $\phi(vw)$, the problem of finding an in-tree that minimizes the total cost $\sum_{vw\in E_\tau}\phi(vw)$ is called the *minimum weight rooted arborescence problem*. We therefore can solve the pricing problem by solving the minimum weight rooted arborescence problem. Note that an arborescence is usually defined as an out-tree, but the direction of the rooted tree does not make any essential difference to this problem.

The minimum weight rooted arborescence problem takes as inputs a directed graph $G=(V,E)$, a root node $r\in V$ and an edge cost function $\phi: E\to\mathbb{R}$. The problem consists of finding a rooted arborescence with minimum total edge cost. The problem can be solved in $O(|E||V|)$ time by Edmonds' algorithm (Edmonds, 1967). Bock (1971) and Chu and Liu (1965) obtained similar results. Gabow et al. (1986) presented the best results so far with an algorithm of time complexity $O(|E|+|V|\log|V|)$, which uses Fibonacci heap.

In each iteration of the in-tree generation phase of our algorithm, an in-tree τ that minimizes the left-hand side of (4) is computed and is added into T provided that it satisfies (4).

3.2 Stopping criteria of the delayed column generation method

In this subsection, we consider the stopping criteria of the delayed column generation method. The following theorem shows that we can obtain an upper bound of $\mathrm{OPT}_{LP(T_{\mathrm{all}})}$ at each iteration of the delayed column generation method.

Theorem 1 *Let $\hat{\lambda}_i\geq 0$ be real numbers for $i\in V$ and $\alpha=\min_{j\in T_{\mathrm{all}}}\left\{\sum_{i\in V}a_{ij}\hat{\lambda}_i\right\}$. If $\alpha>0$ holds, then $\sum_{i\in V}b_i(\hat{\lambda}_i/\alpha)$ is an upper bound of $\mathrm{OPT}_{LP(T_{\mathrm{all}})}$.*

Proof: Let $\mathrm{OPT}_{LP(T_{\mathrm{all}})}$ (resp., $\mathrm{OPT}_{D(T_{\mathrm{all}})}$) be the optimal value of the problem $LP(T_{\mathrm{all}})$ (resp., $D(T_{\mathrm{all}})$). From the duality theorem, $\mathrm{OPT}_{LP(T_{\mathrm{all}})}=\mathrm{OPT}_{D(T_{\mathrm{all}})}$. By the definition of α (>0), $\sum_{i\in V}a_{ij}\hat{\lambda}_i\geq\alpha$ holds for all $j\in T_{\mathrm{all}}$, which is equivalent with

$$\sum_{i\in V}a_{ij}(\hat{\lambda}_i/\alpha)\geq 1,\qquad\qquad \forall j\in T_{\mathrm{all}}.$$

Thus, $(\hat{\lambda}_i/\alpha \mid i\in V)$ is a feasible solution of the problem $D(T_{\mathrm{all}})$ and its objective value $w=\sum_{i\in V}b_i(\hat{\lambda}_i/\alpha)$ satisfies $\mathrm{OPT}_{D(T_{\mathrm{all}})}\leq w$. Hence we have $\mathrm{OPT}_{LP(T_{\mathrm{all}})}\leq w$. □

Theorem 1 implies that we can obtain an upper bound of $\mathrm{OPT}_{P(T_{\mathrm{all}})}$ at each iteration of the delayed column generation method. Furthermore, $\lfloor\sum_{i\in V}b_i(\lambda_i^*/\alpha)\rfloor$ is an upper bound of $\mathrm{OPT}_{P(T_{\mathrm{all}})}$ because $\lfloor\mathrm{OPT}_{LP(T)}\rfloor$ gives an upper bound of $\mathrm{OPT}_{P(T)}$ for any $T\subseteq T_{\mathrm{all}}$. Note that

$$\mathrm{OPT}_{LP(T)}\leq\mathrm{OPT}_{LP(T_{\mathrm{all}})}\leq\frac{\mathrm{OPT}_{LP(T)}}{\alpha}$$

always holds, where $\mathrm{OPT}_{LP(T)}=\mathrm{OPT}_{D(T)}=\sum_{i\in V}b_i\lambda_i^*$. Thus, we can obtain $\lfloor\mathrm{OPT}_{LP(T_{\mathrm{all}})}\rfloor$ even without executing the delayed column generation method until the end (i.e., until there is no tree $\tau\in T_{\mathrm{all}}$ that satisfies (4)), provided that

$$\lfloor\mathrm{OPT}_{LP(T)}/\alpha\rfloor\leq\mathrm{OPT}_{LP(T)}. \tag{6}$$

We employ condition (6) as one of the stopping criteria of our delayed column generation method.

If only condition (6) is employed and $\mathrm{OPT}_{LP(T)}$ is large, then there is a possibility that the delayed column generation method generates a lot of in-trees. Thus, we employ an additional condition

$$\frac{\mathrm{OPT}_{LP(T)}/\alpha-\mathrm{OPT}_{LP(T)}}{\mathrm{OPT}_{LP(T)}}\leq\epsilon,$$

which is equivalent with

$$\alpha\geq\frac{1}{1+\epsilon}, \tag{7}$$

where $\epsilon\geq 0$ is a parameter that represents the accuracy of the obtained upper bound $\mathrm{OPT}_{LP(T)}/\alpha$ against $\mathrm{OPT}_{LP(T_{\mathrm{all}})}$. In the computational experiments in Section 5, we set $\epsilon:=0.0001$. We observed through preliminary computational experiments that even if the delayed column generation method was stopped by conditions (6) or (7), good solutions for $P(T_{\mathrm{all}})$ were obtained, which implies that the set of in-trees generated until one of these conditions is satisfied seems to be sufficient for obtaining high quality solutions to $P(T_{\mathrm{all}})$.

3.3 Initial set of in-trees

The delayed column generation method can be executed even with only one initial in-tree. However, we observed through preliminary experiments that the computation time was usually reduced if an initial set with more in-trees was given. We also noticed that for randomly generated in-trees the computation time did not decrease so much when we increase the number of in-trees in the initial set beyond $|V|$. We therefore employ $|V|$ randomly generated in-trees as the initial set of in-trees.

Imahori et al. (2009) proved that finding one packed in-tree that satisfies the node capacity restriction (2) is NP-hard. Consequently, it is difficult to create an initial set of in-trees for it and hence we dealt only with problems with $t(e)\ll b_i, \forall e\in\delta^+(i)$ and $h(e)\ll b_i, \forall e\in\delta^-(i)$ for all $i\in V$.

3.4 Proposed algorithm to generate in-trees

The algorithm to generate in-trees based on the delayed column generation approach, which we call GenInTrees, is formally described as follows:

Algorithm GenInTrees

Input a graph $G = (V, E)$, a root node $r \in V$, tail and head consumption functions on edges $t : E \to \mathbb{R}_+$, $h : E \to \mathbb{R}_+$, node capacities $b_i \in \mathbb{R}_+$ for all $i \in V$ and a parameter ϵ.

Output a set of in-trees T, a set S of feasible solutions of $P(T)$, an upper bound $\lfloor \mathrm{OPT}_{LP(T)}/\alpha \rfloor$ and a lower bound $\sum_{j \in T} \lfloor x_j^{\max} \rfloor$.

1. Create the initial set T_0 of $|V|$ in-trees randomly. Set $T := T_0$, $S := \emptyset$ and $x_j^{\max} := 0$ for all $j \in T$.
2. Solve the problem $LP(T)$. Let $\mathrm{OPT}_{LP(T)}$ be the optimal value, $(x_j^* \mid j \in T)$ be the obtained optimal solution and $(\lambda_i^* \mid i \in V)$ be the corresponding optimal dual solution. Set $S := S \cup \{(\lfloor x_j^* \rfloor \mid j \in T)\}$.
3. If $\sum_{j \in T} \lfloor x_j^* \rfloor > \sum_{j \in T} \lfloor x_j^{\max} \rfloor$ holds, then set $x_j^{\max} := x_j^*$ for all $j \in T$.
4. After setting the edge costs $\phi(vw) := \lambda_v^* t(vw) + \lambda_w^* h(vw)$ for all $vw \in E$, execute Edmonds' algorithm. Let τ be the in-tree with minimum total cost and α be its cost.
5. If $\lfloor \mathrm{OPT}_{LP(T)}/\alpha \rfloor \leq \mathrm{OPT}_{LP(T)}$ or $\alpha \geq 1/(1+\epsilon)$ holds, then output the set of in-trees T, the upper bound $\lfloor \mathrm{OPT}_{LP(T)}/\alpha \rfloor$, the lower bound $\sum_{j \in T} \lfloor x_j^{\max} \rfloor$ and stop. Else set $T := T \cup \{\tau\}$ and return to Step 2.

The GenInTrees algorithm outputs a set S of feasible solutions of $P(T)$. Although it is not necessarily to keep the set S for executing the GenInTrees algorithm, it contains useful information to be used by the algorithm presented next.

4 In-trees Packing Algorithm

4.1 Evaluation criteria of in-trees

The GenInTrees algorithm could generate in-trees to obtain high quality solutions of $P(T_{\mathrm{all}})$. In this subsection, we propose a greedy algorithm to pack the in-trees enumerated by the GenInTrees algorithm. We use the maximum packing number, calculated based on the available capacity in each node, as the evaluation criterion of each in-tree. Let $(x_j \mid j \in T)$ be the current feasible solution of $P(T_{\mathrm{all}})$. The current available capacity in each node $i \in V$ is defined as

$$\bar{b}_i = b_i - \sum_{j \in T} a_{ij} x_j. \tag{8}$$

Then the maximum packing number Δ_j of each in-tree $j \in T$ is calculated as follows:

$$\Delta_j = \min_{i \in V} \left\lfloor \frac{\bar{b}_i}{a_{ij}} \right\rfloor. \tag{9}$$

In the remainder of this subsection, we focus on one iteration of our greedy algorithm, i.e., which in-tree j is chosen to increase its packing number x_j, and how much x_j is increased.

In each iteration of our greedy algorithm, an in-tree $j \in T$ that maximizes Δ_j is chosen and the value of x_j is increased. Let $j^* \in T$ be an in-tree with the highest Δ_j among all $j \in T$. A simple approach to decide the amount of increment is to use the value of Δ_{j^*} (i.e., $x_{j^*} := x_{j^*} + \Delta_{j^*}$); however, this approach does not give good solutions for $P(T_{\mathrm{all}})$. The available capacities $\bar{b}_i$ on the nodes are decreased as the packing number x_{j^*} of the in-tree $j^* \in T$ is increased, and the amount of decrement of Δ_j is different among in-trees. Thus, we employ an approach in which the in-tree with the highest Δ_j is packed while its Δ_j value is the highest.

For any in-tree $j \in T \setminus \{j^*\}$, we define q_j as the minimum value such that after increasing x_{j^*} by q_j, the resulting Δ_{j^*} becomes smaller than the resulting Δ_j. Such a value of q_j must satisfy the following condition for all $i \in V$:

$$\left\lfloor \frac{\bar{b}_i - a_{ij^*} q_j}{a_{ij}} \right\rfloor > \Delta_{j^*} - q_j. \tag{10}$$

Because we use q_j as the value to increase the packing number x_{j^*} of in-tree j^*, it has to satisfy

$$0 \leq q_j \leq \Delta_{j^*}. \tag{11}$$

The right-hand side of (10) is an integer and hence the condition (10) is equivalent to

$$\frac{\bar{b}_i - a_{ij^*} q_j}{a_{ij}} - 1 \geq \Delta_{j^*} - q_j. \tag{12}$$

Let $\lceil x \rceil$ be the ceiling function of x. Condition (12) is satisfied for all $i \in V$ if and only if

$$\begin{cases} \left\lfloor \dfrac{\bar{b}_i - a_{ij}(\Delta_{j^*} + 1)}{a_{ij^*} - a_{ij}} \right\rfloor \geq q_j, & (a_{ij^*} > a_{ij}), \ (13) \\ \left\lceil \dfrac{a_{ij}(\Delta_{j^*} + 1) - \bar{b}_i}{a_{ij} - a_{ij^*}} \right\rceil \leq q_j, & (a_{ij^*} < a_{ij}), \ (14) \\ \left\lfloor \dfrac{\bar{b}_i}{a_{ij}} \right\rfloor \geq \Delta_{j^*}, & (a_{ij^*} = a_{ij}), \ (15) \end{cases}$$

are satisfied for all $i \in V$. Let $Q_j \subseteq \mathbb{Z}$ be the set of all integer values of q_j that satisfy (11), (13) and (14) for all $i \in V$. If $Q_j = \emptyset$ holds or condition (15) is not satisfied for some $i \in V$, then Δ_j never becomes higher than Δ_{j^*} (in this case, we assume $q_j = +\infty$ for convenience). Otherwise, Δ_j becomes higher than Δ_{j^*} by increasing x_{j^*} by $q_j = \min\{Q_j\}$. Hence, if we increase the value of x_{j^*} by q, where $q = \min_{j \in T \setminus \{j^*\}}\{q_j\}$, Δ_j becomes higher than Δ_{j^*} for some $j \in T \setminus \{j^*\}$.

4.2 Efficient data structure

It is necessary to recalculate Δ_j and q_j for all in-trees whenever an in-tree j^* with the highest Δ_j among all

$j \in T$ is packed. We propose an efficient method to recalculate these values, which eliminates unnecessary recalculation by maintaining a sorted array such that its elements are upper bounds of Δ_j.

Let $\mathfrak{D}$ be an array with $|T|$ elements sorted in non-increasing order and $j_k \in T$ be the index of the tree corresponding to the kth cell of $\mathfrak{D}$. This array is maintained so that it satisfies $\Delta_{j_k} \leq \mathfrak{D}[k]$ for all $k \in \{1, \ldots, |T|\}$ and $\Delta_{j_1} = \mathfrak{D}[1]$. Then the in-tree j_1 has the highest Δ_j among all $j \in T$.

The algorithm calculates the values of q_{j_k} for $k = 2, 3, \ldots$ in this order, by conditions (11), (13), (14) and (15). Assume that the values of q_{j_k} are calculated until the element in the k'th position. We define $q^{\min}_{j_{k'}}$ as the minimum value of q_{j_k} among the ones calculated, i.e., $q^{\min}_{j_{k'}} = \min_{k \in [2,k']} q_{j_k}$. Hence, if we increase the value of x_{j_1} by $q^{\min}_{j_{k'}}$, there is at lease one in-tree whose Δ_{j_k} becomes higher than Δ_{j_1} among the ones calculated (provided that $q^{\min}_{j_{k'}}$ is finite). Then, suppose

$$\mathfrak{D}[k'+1] \leq \Delta_{j_1} - q^{\min}_{j_{k'}}. \tag{16}$$

Because $\mathfrak{D}$ is sorted, $\mathfrak{D}[k] \leq \Delta_{j_1} - q^{\min}_{j_{k'}}$ holds for all $k \geq k'+1$. By definition, $\Delta_{j_k} \leq \mathfrak{D}[k]$ holds for all $k \in \{1, \ldots, |T|\}$ and the value of Δ_{j_k} never increases for all k when x_{j_1} is increased. Hence, for all $k \geq k'+1$, Δ_{j_k} does not exceed Δ_{j_1} unless the increment in x_{j_1} is bigger then $q^{\min}_{j_{k'}}$, which implies $q_{j_k} > q^{\min}_{j_{k'}}$. Therefore, if condition (16) is satisfied, $q^{\min}_{j_{k'}} = q = \min_{j \in T \setminus \{j^*\}} q_j$ holds, and hence it is not necessary to calculate q_{j_k} for all $k \geq k'+1$. Then the value of x_{j_1} is increased by q and the values of $\bar{b}_i$ are updated for all $i \in V$ by recomputing them by (8).

We can reduce the computational effort to update Δ_{j_k} for all $k \in \{1, \ldots, |T|\}$ by using a similar idea to the one for calculating q. For $k = 1, 2, \ldots$ in this order, the algorithm calculates Δ_{j_k} by (9). We denote the new value of Δ_{j_k} by $\bar{\Delta}_{j_k}$. Assume that the values of $\bar{\Delta}_{j_k}$ are calculated until the element in the k''th position. We define $\bar{\Delta}^{\min}_{j_{k''}}$ as the minimum value of $\bar{\Delta}_{j_k}$ among the ones calculated, i.e., $\bar{\Delta}^{\min}_{j_{k''}} = \min_{k \in [1,k'']} \bar{\Delta}_{j_k}$. Suppose

$$\mathfrak{D}[k''+1] < \bar{\Delta}^{\min}_{j_{k''}}. \tag{17}$$

Because $\mathfrak{D}$ is sorted, $\mathfrak{D}[k] < \bar{\Delta}^{\min}_{j_{k''}}$ holds for all $k \geq k''+1$. Then the algorithm stops recalculating Δ_{j_k}, and for $k = 1, 2, \ldots, k''$, it updates $\mathfrak{D}[k]$ with the recomputed value $\bar{\Delta}_{j_k}$ (i.e., $\mathfrak{D}[k] := \bar{\Delta}_{j_k}$). Afterwards, it sorts the first k'' elements of $\mathfrak{D}$ in the non-increasing order. The resulting array $\mathfrak{D}$ is sorted in the whole range (i.e., $\mathfrak{D}[1] \geq \mathfrak{D}[2] \geq \cdots \geq \mathfrak{D}[|T|]$ holds), and it satisfies $\bar{\Delta}_{j_k} \leq \mathfrak{D}[k]$ for all $k \in \{1, \ldots, |T|\}$ and $\bar{\Delta}_{j_1} = \mathfrak{D}[1]$, since we have $\bar{\Delta}_{j_k} \leq \Delta_{j_k}$.

4.3 Proposed algorithm to pack in-trees

This section summarizes the greedy algorithm proposed in Section 4.1, together with the data structure in Section 4.2. We call the algorithm PackInTrees, which is formally described as follows.

Algorithm PackInTrees

Input a problem instance of $P(T)$ and a feasible solution $(x^{(0)}_j \mid j \in T)$ of $P(T)$.
Output a feasible solution $(x_j \mid j \in T)$ of $P(T)$.

1. Set $x_j := x^{(0)}_j$ for all $j \in T$ and calculate available capacities $\bar{b}_i$ for all $i \in V$ by (8).
2. Calculate the evaluation criteria Δ_j for all $j \in T$ by 8 and set $\mathfrak{D}[j] := \Delta_j$.
3. Sort $\mathfrak{D}$ in the non-increasing order, and let $j_k \in T$ be the index of the tree corresponding to the kth cell of $\mathfrak{D}$, i.e., $\mathfrak{D}[k] = \Delta_{j_k}$ holds for all $k \in \{1, \ldots, |T|\}$, and $\Delta_{j_1} \geq \Delta_{j_2} \geq \cdots \geq \Delta_{j_{|T|}}$ is satisfied.
4. Set $q^{\min} := \mathfrak{D}[1]$ and $k := 2$.
5. Let $Q_{j_k} \subseteq \mathbb{Z}$ be the set of all integer values of q_{j_k} that satisfy (11), (13) and (14) with $j = j_k$ for all $i \in V$. If $Q_{j_k} \neq \emptyset$ and $q_{j_k} = \min\{Q_{j_k}\} < q^{\min}$ are satisfied and (15) holds for all $i \in V$, then set $q^{\min} := q_{j_k}$.
6. If $k = |T|$, then go to Step 8.
7. If $\mathfrak{D}[k+1] > \mathfrak{D}[1] - q^{\min}$, then set $k := k+1$ and return to Step 5.
8. Set $x_{j_1} := x_{j_1} + q^{\min}$. Recalculate available capacities $\bar{b}_i$ for all $i \in V$ by 8.
9. Set $\Delta^{\min} := \mathfrak{D}[1]$ and $k := 1$.
10. Recalculate Δ_{j_k} by (9) and set $\mathfrak{D}[k] := \Delta_{j_k}$. If $\Delta_{j_k} < \Delta^{\min}$, then set $\Delta^{\min} := \Delta_{j_k}$.
11. If $k = |T|$, then go to Step 13.
12. If $\mathfrak{D}[k+1] \geq \Delta^{\min}$, then set $k := k+1$ and return to Step 10.
13. Sort the first k elements of $\mathfrak{D}$ in the non-increasing order, and modify $j_{\hat{k}}$ ($\hat{k} \in \{1, \ldots, k\}$) accordingly.
14. If $\mathfrak{D}[1] = 0$, then output the feasible solution $(x_j \mid j \in T)$ and stop; otherwise, return to Step 4.

One iteration of the PackInTrees algorithm consists of calculating $q^{\min}$ and $\Delta^{\min}$ and sorting $\mathfrak{D}$. Its worst case time complexity is $O(|V||T| + |T| \log |T|)$. Let k' be the value of k in Step 8 and k'' be the value of k in Step 13. It is not hard to show that $k'' \geq k'$ holds, and by using these parameters, the actual time complexity becomes $O(|V|(k'+k'') + k'' \log k'') = O(k''(|V| + \log k''))$, which is usually much smaller than the worst-case complexity because $k'' \ll |T|$ holds in many cases. In each iteration, at least one in-tree is packed and hence the maximum number of iterations is $\mathrm{OPT}_{LP(T_{\mathrm{all}})}$. Hence, the whole algorithm runs in $O(\mathrm{OPT}_{LP(T_{\mathrm{all}})}(|V||T| + |T| \log |T|))$ time.

We set an initial feasible solution $(x^{(0)}_j \mid j \in T)$ as an input for the PackInTrees algorithm. We observed through preliminary experiments that if we set $x^{(0)}_j := 0$ for all $j \in T$, the PackInTrees algorithm does not output good solutions. On the other

hand, good solutions are found by setting some of the solutions obtained by the GenInTrees algorithm, $(\lfloor x^*_j \rfloor \mid j \in T) \in S$, as the initial feasible solution of PackInTrees algorithm. Let ℓ be the number of times PackInTrees algorithm is executed. We employ the ℓ solutions with largest objective values. In case of ties, we prefer the solutions generated later by the GenInTrees algorithm.

5 Computational Experiments

5.1 Problem instances

We use two types of instances in our experiment. The first one is based on sensor location data used by Heinzelman et al. (2002) and Sasaki et al. (2007) in their papers about sensor networks. From their data, we generated complete graphs with symmetric tail and head consumption functions and node capacities, where the consumption functions are equivalent to the amount of energy consumed to transmit and receive packets, and node capacities are equivalent to the capacities of sensor batteries in their papers. We call the instances hcb100, sfis100-1, sfis100-2 and sfis100-3, where hcb100 is the instance generated using the sensor location data in Heinzelman et al. (2002), and sfis100-1, 2 and 3 are the instances generated using the sensor location data called data1, 2 and 3, respectively, in Sasaki et al. (2007).

The second type consists of random graphs whose out-degrees are distributed in a small range. We named them as "rnd$n$-$d_{\min}$-$d_{\max}$," where n is the number of nodes, $d_{\min}$ is the minimum out-degree, and $d_{\max}$ is the maximum out-degree. Three problem instances were generated, which are rnd100-5-10, rnd100-30-50 and rnd200-5-10. Tail and head consumptions for these instances were randomly chosen from the integers in the intervals $[10, 50]$ and $[1, 5]$, respectively, for all edges except that the tail consumption of edges entering the root node r were randomly chosen from the integers in the intervals $[100, 500]$ so that these edges cannot be used frequently. Node capacities were set to 100,000 for all $i \in V \setminus \{r\}$ and $+\infty$ for the root node r.

The algorithms were coded in the C++ language and ran on a Dell Precision 470 (Xeon (NetBurst) 3GHz, 2MB cache, 1GB memory). We used the primal simplex method in GLPK4.34* as LP solver.

5.2 Experimental results

Figure 1 represents the behavior of GenInTrees algorithm applied to sfis100-1. The figure shows the improvement of $\mathrm{OPT}_{LP(T)}$ and of the best values of the upper and lower bounds of $P(T_{\mathrm{all}})$ as in-trees are added to T in each iteration. Along with this improvement,

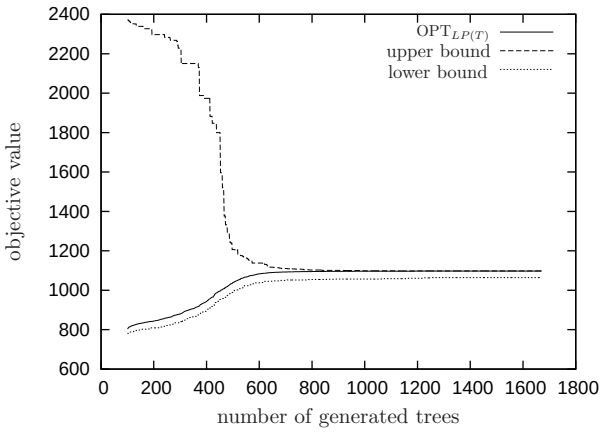

Figure 1. Behavior of the GenInTrees algorithm applied to sfis100-1

the difference between the upper and lower bounds becomes smaller and the ratios of improvement decrease. In general, this tendency is often observed when applying the delayed column generation method. After a certain number of iterations, we can affirm that we obtained good upper and lower bounds.

The solutions output by the GenInTrees algorithm are used as the initial solutions for the PackInTrees algorithm. Figure 2 shows the relationship between the objective values of the solutions output by the GenInTrees algorithm (horizontal axis) and the objective values of the solutions output by the PackInTrees algorithm (vertical axis) applied to sfis100-1. From the figure, we can notice a strong correlation between the values. This tendency was also observed for other instances. Thus, it is not necessary to input the solutions of GenInTrees with small objective values into the PackInTrees algorithm, and for this reason, we use only a percentage of solutions with largest objective values for the initial solutions of PackInTrees.

Table 1 shows the results of the proposed algorithm with $\ell = 1$, $\lfloor 0.01|T| \rfloor$, $\lfloor 0.05|T| \rfloor$ and $|T|$ for the problem instances explained in Section 5.1. The first three columns represent instance names, number of nodes (without the root node) $|V \setminus \{r\}|$, and number of edges $|E|$. Column $|T|$ shows the number of in-trees generated by algorithm GenInTrees, and columns UB and LB show the upper and lower bounds of $\mathrm{OPT}_{P(T_{\mathrm{all}})}$, respectively, computed by GenInTrees. The following columns represent objective values, denoted "Obj.," and computation times in seconds of the proposed algorithm for the four values of ℓ. Comparing the results for $\ell = 1$ and $\ell = \lfloor 0.01|T| \rfloor$, we can notice that some objective values improve. From $\ell = \lfloor 0.01|T| \rfloor$ to $\ell = \lfloor 0.05|T| \rfloor$, only the objective value of sfis100-1 improves. Moreover, when we increase the value of ℓ over $\lfloor 0.05|T| \rfloor$, the objective value does not improve. These

*GLPK-GNU Project-Free Software Foundation (FSF), `http://www.gnu.org/software/glpk/`, 20, April, 2009.

Table 1. Influence of parameter ℓ on the results of the proposed algorithm

Instance name	$\|V \setminus \{r\}\|$	$\|E\|$	GenInTrees			$\ell = 1$		$\ell = \lfloor 0.01\|T\| \rfloor$		$\ell = \lfloor 0.05\|T\| \rfloor$		$\ell = \|T\|$	
			$\|T\|$	UB	LB	Obj.	Time (s)	Obj.	Time (s)	Obj.	Time (s)	Obj.	Time (s)
hcb100	100	10,100	2752	1124	1095	1121	210	1121	213	1121	230	1121	651
sfis100-1	100	10,100	1669	1097	1064	1089	83	1090	85	1092	95	1092	293
sfis100-2	100	10,100	1700	1097	1065	1090	87	1090	90	1090	96	1090	307
sfis100-3	100	10,100	1378	1101	1067	1095	59	1095	60	1095	65	1095	171
rnd100-5-10	100	776	940	3225	3186	3210	31	3212	31	3212	33	3212	76
rnd100-30-50	100	3991	1985	6250	6209	6235	99	6235	99	6235	106	6235	201
rnd200-5-10	200	1525	2237	2702	2631	2677	1011	2681	1015	2681	1032	2681	1701

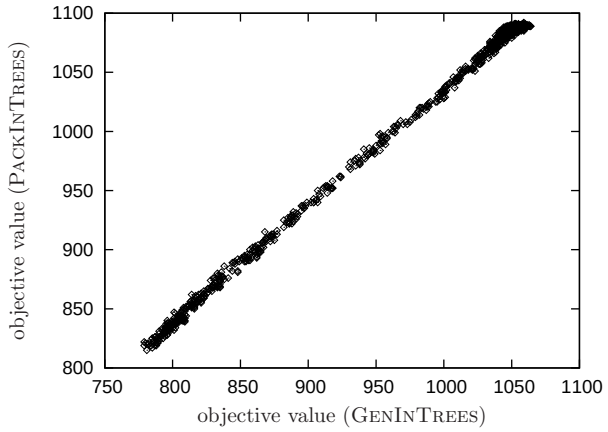

Figure 2. Relationship between objective values of the input (given by GenInTrees) and output solutions for PackInTrees applied to sfis100-1

results are consistent with the ones observed from Figure 2. The computation time increases about 2–3 times if we compare $\ell = 1$ and $\ell = |T|$, and about 10% if we compare $\ell = 1$ and $\ell = \lfloor 0.05|T| \rfloor$. The proposed algorithm is not sensitive to the value of ℓ and it is not necessary to make an effort to tune it. Thus, we set $\ell := \lfloor 0.05|T| \rfloor$.

We then compare the solutions obtained by the proposed algorithm with the ones obtained by the algorithm in Sasaki et al. (2007). Note that their algorithm keeps sending packets even though the base station does not receive packets from all sensors, where this situation happens only if there exists at least one sensor whose battery has run out, and they reported the number of times packets are sent (which they call rounds) for several values of the number of available sensors. Among such results, we use the number of rounds when all sensors are available, because in this paper we consider the number of spanning in-trees, which corresponds to the number of times packets are sent from all the sensors.

Table 2 compares the solutions obtained by the proposed algorithm with the ones obtained by the algorithm in Sasaki et al. (2007). The first six columns are equivalent to the first six columns of Table 1. The next four columns show the results of the proposed algorithm with $\ell = \lfloor 0.05|T| \rfloor$: the objective values, the gaps in % between UB and Obj., i.e., $((\text{UB} - \text{Obj.})/\text{UB}) \times 100$, the numbers of in-trees used in the best solutions, i.e., $|T_{\text{pos}}(x_{\text{best}})|$, where $T_{\text{pos}}(x) = \{j \in T \mid x_j > 0\}$ and x_{best} represents the best solution obtained by the proposed algorithm, and computation times in seconds. The number of rounds reported in Sasaki et al. (2007), which are equivalent to the number of packed in-trees, is also shown for comparison purposes, where a mark "–" means that the result is not available.

The results presented in Table 2 show that our algorithm obtains better results than Sasaki et al. (2007). The gaps between upper bounds and objective values are quite small, indicating that the obtained solutions are close to $\text{OPT}_{P(T_{\text{all}})}$. We can also observe that the number of in-trees used in the best solution x_{best} is always smaller than the number of nodes, ranging around 70–80% of $|V|$ except for two instances. Instance rnd200-5-10 is the only one with 200 nodes and although the number of edges is not much bigger than other instances, the computation time is at least 10 times bigger except for hcb100. Thus, the computational effort increases rapidly when the number of nodes increases. One of the reasons for this behavior is the increase of the computational effort of the LP solver.

6 Conclusions

In this paper, we proposed a two-phase heuristic algorithm for the node capacitated in-tree packing problem. In the first phase, it generates candidate in-trees to be packed employing the delayed column generation method for the LP-relaxation of the problem. We showed that solving the pricing problem is equivalent to solving the minimum weight rooted arborescence problem. In the second phase, the algorithm computes the packing number of each in-tree by first modifying feasible solutions of the LP-relaxation problem and then improving them with a greedy algorithm. We proposed an efficient data structure that makes use of the properties of the evaluation criteria. The proposed algorithm obtains solutions that are close to the upper bounds and is proved to be effective for this problem.

Table 2. Computational results

Instance name	$\|V \setminus \{r\}\|$	$\|E\|$	GenInTrees			Proposed Algorithm ($\ell = \lfloor 0.05\|T\| \rfloor$)				Sasaki et al. (2007)
			$\|T\|$	UB	LB	Obj.	Gap (%)	$\|T_{\mathrm{pos}}(x_{\mathrm{best}})\|$	Time (s)	
hcb100	100	10,100	2752	1124	1095	1121	0.27	69	230	–
sfis100-1	100	10,100	1669	1097	1064	1092	0.46	77	95	961
sfis100-2	100	10,100	1700	1097	1065	1090	0.64	74	96	969
sfis100-3	100	10,100	1378	1101	1067	1095	0.54	75	65	1022
rnd100-5-10	100	776	940	3225	3186	3212	0.40	98	33	–
rnd100-30-50	100	3991	1985	6250	6209	6235	0.24	95	106	–
rnd200-5-10	200	1525	2237	2702	2631	2681	0.78	154	1032	–

Acknowledgment

The authors would like to thank Celso Satoshi Sakuraba for his detailed comments on earlier versions of this paper.

References

Bock, F. C. (1971). An algorithm to construct a minimum directed spanning tree in a directed network. In Avi-Itzak, B., editor, *Developments in Operations Research*, pages 29–44. Gordon and Breach, New York.

Calinescu, G., Kapoor, S., Olshevsky, A., and Zelikovsky, A. (2003). Network lifetime and power assignment in ad-hoc wireless networks. In *Proceedings of the 11th European Symposium on Algorithms*, volume 2832 of *Lectre Notes in Computer Science*, pages 114–126. Springer.

Chu, Y. and Liu, T. (1965). On the shortest arborescence of a directed graph. *Science Sinica*, 14:1396–1400.

Edmonds, J. (1967). Optimum branchings. *Journal of Research of the National Bureau of Standards*, 71B:233–240.

Edmonds, J. (1973). Edge-disjoint branchings. In Rustin, B., editor, *Combinatorial Algorithms*, pages 91–96. Academic Press.

Gabow, H. N., Galil, Z., Spencer, T., and Tarjan, R. E. (1986). Efficient algorithms for finding minimum spanning trees in undirected and directed graphs. *Combinatorica Archive*, 6:109–122.

Gabow, H. N. and Manu, K. S. (1998). Packing algorithms for arborescences (and spanning trees) in capacitated graphs. *Mathematical Programming*, 82:83–109.

Heinzelman, W. B., Chandrakasan, A. P., and Balakrishnan, H. (2002). An application-specific protocol archtiecture for wireless microsensor networks. *IEEE Transactions on Wireless Communications*, 1:660–670.

Imahori, S., Miyamoto, Y., Hashimoto, H., Kobayashi, Y., Sasaki, M., and Yagiura, M. (2009). The complexity of the node capacitated in-tree packing problem. In *Proceedings of International Network Optimization Conference*.

Korte, B. and Vygen, J. (2007). *Combinatorial Optimization: Theory and Algorithms*. Springer-Verlag, 4th edition.

Mader, W. (1981). On n-edge-connected digraphs. *Combinatorica*, 1:385–386.

Sasaki, M., Furuta, T., Ishizaki, F., and Suzuki, A. (2007). Multi-round topology construction in wireless sensor networks. In *Proceedings of the Asia-Pacific Symposium on Queueing Theory and Network Applications*, pages 377–384.

Schrijver, A. (2003). *Combinatorial Optimization*. Springer-Verlag.