Journal of the Operations Research Society of Japan
Vol. 54, No. 4, December 2011, pp. 219–236
© The Operations Research Society of Japan

LAGRANGIAN-BASED COLUMN GENERATION FOR THE NODE CAPACITATED IN-TREE PACKING PROBLEM

Yuma Tanaka　　Shinji Imahori　　Mutsunori Yagiura
Nagoya University

(Received August 1, 2011; Revised September 30, 2011)

Abstract In this paper, we deal with 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 is to find a subset of rooted spanning in-trees and their packing numbers, where the packing number of an in-tree is the number of times it is packed, so as to maximize the sum of packing numbers under the constraint that the total consumption of the packed in-trees at each node does not exceed the capacity of the node. This problem is known to be NP-hard.

Previously, we proposed a two-phase heuristic algorithm for this problem. The algorithm generates promising candidate in-trees to be packed in the first phase and computes the packing number of each in-tree in the second phase. In this paper, we improve the first phase algorithm by using Lagrangian relaxation instead of LP (linear programming) relaxation.

We conducted computational experiments on graphs used in related papers and on randomly generated instances. The results indicate that our new algorithm generates in-trees faster than our previous algorithm and obtains better solutions than existing algorithms without generating many in-trees.

Keywords: Combinatorial optimization, Lagrangian relaxation, LP relaxation, column generation, sensor network

1. Introduction

In this paper, we consider the *node capacitated in-tree packing problem* (NCIPP). The input consists of a directed graph, a root node, a node capacity function and edge consumption functions for heads and tails. The problem is to find a subset of rooted spanning in-trees and their packing numbers, where the packing number of an in-tree is the number of times it is packed, so as to maximize the sum of packing numbers under the constraint that the total consumption of the packed 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_{all} as the set of all spanning 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 $j \in T_{\text{all}}$ leaving (resp., entering) a node $i \in V$. The consumption a_{ij} of an in-tree $j \in T_{\text{all}}$ at a node $i \in V$ is defined as

$$a_{ij} = \sum_{e \in \delta_j^+(i)} t(e) + \sum_{e \in \delta_j^-(i)} h(e). \tag{1.1}$$

We call the first term of this equation (1.1) *tail consumption*, and the second term *head consumption*. The node capacitated in-tree packing problem is to find a subset $T \subseteq T_{\text{all}}$

of spanning in-trees and the packing number x_j of each in-tree $j \in T$ subject to the node capacity restriction

$$\sum_{j \in T} a_{ij} x_j \leq b_i, \quad \forall i \in V, \tag{1.2}$$

so as to maximize the total number of packed in-trees $\sum_{j \in T} x_j$. Throughout this paper, an in-tree means a spanning in-tree even if we do not clearly state spanning.

This problem is known to be NP-hard [12]. 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 [3, 10, 17]. 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. [10] proposed an algorithm, called LEACH-C (low energy adaptive clustering hierarchy centralized), that uses arborescences with limited height for communication topologies. For more energy efficient communication networks, a multiround topology construction problem was formulated as an integer programming problem, and a heuristic solution method was proposed in [17]. In the formulation of [3], 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. [3] 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 has been investigated from long ago [6], and it is known that the problem is solvable in polynomial time [14]. Its capacitated version is also solvable in polynomial time [9, 15, 16]. Furthermore, a fundamental result in [6] was recently generalized in [13].

For the node capacitated in-tree packing problem, we proposed a two-phase algorithm [18]. 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 column generation method for the LP (linear programming) relaxation of the problem to generate promising candidate 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.

In this paper, we propose a new first-phase algorithm. The new algorithm employs the Lagrangian relaxation instead of the LP relaxation, and it uses the subgradient method to

Copyright © by ORSJ. Unauthorized reproduction of this article is prohibited.

obtain a good Lagrangian multiplier vector. One of the merits of the classical subgradient method is that it is simple and easy to implement; however, it was rather slow and took long time to generate sufficient number of in-trees. To alleviate this, we incorporate various ideas to speed up the algorithm, e.g., rules to decrease the number of in-trees used by the subgradient method, and to reduce the practical computation time for each iteration of the subgradient method.

We conducted computational experiments on graphs used in related papers and on randomly generated instances with up to 200 nodes. The results show that the new algorithm obtains solutions whose gaps to the upper bounds are quite small, and comparisons with existing algorithms show that our new method works more efficiently than them.

2. Formulation

The node capacitated in-tree packing problem (NCIPP) can be formulated as the following IP problem:

$$\begin{aligned} \text{maximize} \quad & \sum_{j \in T_{\text{all}}} x_j, \\ \text{subject to} \quad & \sum_{j \in T_{\text{all}}} a_{ij} x_j \leq b_i, \quad \forall i \in V, \\ & x_j \geq 0, \qquad x_j \in \mathbb{Z}, \ \forall j \in T_{\text{all}}. \end{aligned} \tag{2.1}$$

The notations are summarized as follows:

- V: the set of nodes,
- T_{all}: the set of all in-trees rooted at the given root $r \in V$,
- a_{ij}: the consumption (defined by equation (1.1)) of an in-tree $j \in T_{\text{all}}$ 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 $j \in T_{\text{all}}$,
- $\mathbb{Z}$: the set of all integers.

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

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

If $T = T_{\text{all}}$, the problem $P(T_{\text{all}})$ is equivalent to the original problem (2.1). We denote the optimal value of $P(T)$ by $\text{OPT}_{P(T)}$.

To consider the Lagrangian relaxation problem of $P(T)$, the maximum packing number u_j of each in-tree $j \in T$ is defined as $u_j = \min_{i \in V:\, a_{ij} > 0} \lfloor b_i / a_{ij} \rfloor$ (where $\lfloor y \rfloor$ stands for the

Copyright © by ORSJ. Unauthorized reproduction of this article is prohibited.

floor function of y). The Lagrangian relaxation problem is formally described as follows:

$$LR(T,\lambda) \qquad \text{maximize} \qquad \sum_{j\in T} x_j + \sum_{i\in V} \lambda_i \left(b_i - \sum_{j\in T} a_{ij}x_j \right) \tag{2.2}$$

$$= \sum_{j\in T} c_j(\lambda)x_j + \sum_{i\in V} \lambda_i b_i, \tag{2.3}$$

$$\text{subject to} \quad 0 \le x_j \le u_j, \qquad \forall j \in T,$$

where $\lambda_i \ge 0$ is the Lagrangian multiplier for a node $i \in V$, $\lambda = (\lambda_i \mid i \in V)$ is the vector of Lagrangian multipliers, and $c_j(\lambda) = 1 - \sum_{i\in V} a_{ij}\lambda_i$ is the relative cost of an in-tree $j \in T$. We denote the optimal value of $LR(T,\lambda)$ by $\mathrm{OPT}_{LR(T,\lambda)}$ and an optimal solution of $LR(T,\lambda)$ by $x(\lambda)$. For any $\lambda \ge 0$, an optimal solution $x(\lambda)$ can be calculated easily as follows:

$$x_j(\lambda) = \begin{cases} u_j, & (c_j(\lambda) > 0), \\ 0, & (c_j(\lambda) \le 0), \end{cases} \quad \forall j \in T. \tag{2.4}$$

In general, $\mathrm{OPT}_{LR(T,\lambda)}$ gives an upper bound of $\mathrm{OPT}_{P(T)}$ for any $\lambda \ge 0$.

3. New In-Trees Generating Algorithm

In this section, we explain the new algorithm to generate in-trees. Our algorithm prepares an initial set of in-trees by a simple algorithm in Section 3.1. It then generates in-trees by using the information from Lagrangian relaxation, whose details are explained in Sections 3.2–3.4. To obtain a good upper bound and a Lagrangian multiplier vector, it applies the subgradient method to a current in-tree set, and then it tries to add a new in-tree to the current in-tree set by solving a pricing problem. After adding a new in-tree, it applies the subgradient method to the new in-tree set, and the above steps are repeated until a stopping criterion is satisfied. We also explain a method that obtains good feasible solutions of $P(T_{\mathrm{all}})$ (i.e., this method corresponds to the second-phase algorithm in our previous paper [18]).

3.1. Initial set of in-trees

The 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 observed that, for randomly generated in-trees, the computation time did not decrease much when we increased the number of in-trees in the initial set beyond $|V|$. Based on these observations, we use $|V|$ randomly generated in-trees as the initial set of in-trees.

Remark. Imahori at el. [12] proved that finding one packed in-tree that satisfies the node capacity restriction (1.2) is NP-hard. Consequently, it is not always easy to create an initial set of in-trees for it, and hence we only deal with problem instances for which this part is easy, e.g., those such that any in-tree can be packed at least once, in other words, instances for which $a_{ij} \le b_i$ holds for all $i \in V$ and $j \in T_{\mathrm{all}}$. This condition is satisfied, for example, if $\max_{e\in\delta^+(i)} t(e) + \sum_{e\in\delta^-(i)} h(e) \le b_i$ holds for all $i \in V$, where $\delta^+(i)$ (resp., $\delta^-(i)$) signifies the set of edges in E leaving (resp., entering) each node $i \in V$. This intuitively means that head and tail consumptions are sufficiently small compared to node capacities. Even with this restriction, the problem remains NP-hard as proved by Theorem 7 in [12]. There are many applications in which it is natural to assume that head and tail consumptions are sufficiently small compared to node capacities. For example, in sensor network applications [10, 17], head consumption corresponds to energy consumption for processing received messages,

Copyright © by ORSJ. Unauthorized reproduction of this article is prohibited.

and it is much smaller than tail consumption that corresponds to energy consumption for transmitting messages. Node capacity corresponds to the battery capacity of a sensor, which is usually sufficiently large for transmitting messages thousands of times.

3.2. Subgradient method

We employ the subgradient method to obtain Lagrangian multiplier vectors λ that give good upper bounds of $P(T)$ (i.e., $\mathrm{OPT}_{LR(T,\lambda)}$) for the current set of in-trees T. The subgradient method is a well-known heuristic approach to find a near optimal Lagrangian multiplier vector [1, 7, 11]. It uses the subgradient $s(\lambda) = (s_i \mid i \in V)$, associated with a given λ, defined by $s_i(\lambda) = b_i - \sum_{j\in T} a_{ij}x_j(\lambda)$ for all $i \in V$. This method repeatedly updates a Lagrangian multiplier vector, starting from a given initial vector, by the following formula:

$$\lambda_i := \max\left(0, \lambda_i - \pi\frac{\mathrm{UB}(\lambda) - \mathrm{LB}}{\sum_{i\in V}\{s_i(\lambda)\}^2}s_i(\lambda)\right), \quad \forall i \in V, \tag{3.1}$$

where $\mathrm{UB}(\lambda) = \mathrm{OPT}_{LR(T,\lambda)}$ is an upper bound of $P(T)$, LB is a lower bound of $P(T)$, and $\pi \geq 0$ is a parameter to adjust the step size. We denote $\theta(\lambda) := \pi(\mathrm{UB}(\lambda)-\mathrm{LB})/(\sum_{i\in V}\{s_i(\lambda)\}^2)$, which is called the step size in general. The parameter π is initially set to the value given to the subgradient method. It is then updated after every N iterations by the following rule: The parameter π is halved whenever the best upper bound has not been changed during the last N iterations, where N is a parameter that we set $N = 30$ in our computational experiments. The iteration of the subgradient method is stopped when π becomes less than 0.005. In our algorithm, the above rule to update λ is slightly modified as follows: In the execution of (3.1), we use $s'_i(\lambda)$ instead of $s_i(\lambda)$, where $s'_i(\lambda) = 0$ if $\lambda_i = 0$ and $s_i(\lambda) < 0$ hold immediately before the execution of (3.1), and $s'_i(\lambda) = s_i(\lambda)$ otherwise.

Let $\textsc{SubOpt}(T, \mathrm{LB}, \lambda, \pi)$ be the subgradient method using a lower bound LB, starting from an initial vector λ and a parameter π for an in-tree set T. The procedure SubOpt returns ρ pairs $(\lambda^{(1)}, \pi^{(1)}), \ldots, (\lambda^{(\rho)}, \pi^{(\rho)})$ of Lagrangian multiplier vectors λ and parameters π such that for $k = 1, \ldots, \rho$, the multiplier vector $\lambda^{(k)}$ attains the kth best upper bound $\mathrm{UB}(\lambda)$ among those generated during the search, and the parameter $\pi^{(k)}$ is the value of π when $\lambda^{(k)}$ is found, where the parameter ρ specifies the number of pairs output by SubOpt. These pairs are used in the column generation method whose details are explained in the next section. Refer to the pseudo code for the details of SubOpt, in which $\mathrm{UB}_{\mathrm{best}}$ indicates the best (i.e., minimum) upper bound found during the iteration of SubOpt.

We set $\lambda_i = 1/\min_{j\in T}\sum_{v\in V} a_{vj}$ for all $i \in V$ as the initial Lagrangian multiplier vector and $\pi = 2$ as the initial parameter to adjust the step size if SubOpt is applied to the initial set of in-trees. This simple initial setting of parameters is adopted because it is used only for the first call to SubOpt and does not have much effect on the performance of the algorithm. For the second call or later (i.e., when SubOpt is applied to an in-tree set after adding a new in-tree by the column generation method), the algorithm uses the information of the last execution of SubOpt as follows: The initial values of λ and π are set to $\lambda = \lambda^{(k)}$ and $\pi = \pi^{(k)}$ for the k such that the pair $(\lambda^{(k)}, \pi^{(k)})$ was used to generate the latest new in-tree by the column generation method. With this approach, SubOpt is able to decrease the number of iterations until a good Lagrangian multiplier vector is obtained.

We employ the greedy algorithm PackInTrees proposed in our previous work [18] as a method for producing a lower bound LB (feasible solution) of $P(T)$. This algorithm uses the maximum packing number, calculated based on the available capacity in each node, as the evaluation criterion of each in-tree. The proposed algorithm does not frequently update LB; PackInTrees is applied to an initial in-tree set, and then it is applied whenever a hundred

Copyright © by ORSJ. Unauthorized reproduction of this article is prohibited.

Algorithm SubOpt(T, LB, λ, π)

Input: a set of in-trees T, a lower bound LB of $P(T)$, an initial Lagrangian multiplier vector λ and an initial parameter π.

Output: ρ pairs $(\lambda^{(1)}, \pi^{(1)}), \ldots, (\lambda^{(\rho)}, \pi^{(\rho)})$ such that for $k = 1, \ldots, \rho$, $\lambda^{(k)}$ attains the kth best upper bound among those generated during the iteration, and $\pi^{(k)}$ is the value of π when $\lambda^{(k)}$ is found.

1: Let $\text{UB}_{\text{best}} := +\infty$ and $\Lambda := \emptyset$.
2: Repeat Line 3 to Line 10 N times.
3: Calculate the optimal value $\text{OPT}_{LR(T,\lambda)}$ and an optimal solution $x(\lambda)$ of $LR(T, \lambda)$, and set $\text{UB} := \text{OPT}_{LR(T,\lambda)}$.
4: If $|\Lambda| < \rho$, then let $\Lambda := \Lambda \cap (\lambda, \pi)$ and go to Line 6.
5: If UB is less than the ρth best upper bound attained by $(\lambda^{(\rho)}, \pi^{(\rho)})$ in Λ, then let $\Lambda := \Lambda \setminus (\lambda^{(\rho)}, \pi^{(\rho)}) \cap (\lambda, \pi)$.
6: If $\text{UB}_{\text{best}} > \text{UB}$, then update the best upper bound by $\text{UB}_{\text{best}} := \text{UB}$.
7: Calculate the subgradient by $s_i := b_i - \sum_{j \in T} a_{ij} x_j(\lambda)$ for all $i \in V$.
8: For every $i \in V$, if $\lambda_i = 0$ and $s_i < 0$, then let $s_i := 0$.
9: Calculate the step size $\theta := \pi(\text{UB} - \text{LB})/(\sum_{i \in V} s_i^2)$.
10: Update the Lagrangian multiplier vector by $\lambda_i := \max(0, \lambda_i - \theta s_i)$ for all $i \in V$.
11: If UB_{best} is not updated during the last N iterations, then let $\pi := \pi/2$.
12: If $\pi < 0.005$, then output ρ pairs $(\lambda^{(1)}, \pi^{(1)}), \ldots, (\lambda^{(\rho)}, \pi^{(\rho)})$ in Λ and stop.
13: Return to Line 2.

new in-trees are added, because we confirmed through preliminary experiments that the performance of our algorithm was not affected much by the quality of lower bounds.

The above explanation of the algorithm describes only basic parts, but we also incorporated various ideas to speed up the algorithm, e.g., rules to decrease the number of in-trees used by the subgradient method, and to reduce the practical computation time for each iteration of the subgradient method. The details of these ideas are explained in Section 3.6.

Remark: Without loss of generality, we can assume $b_i = b$ for all $i \in V$, where b is an arbitrary positive constant, e.g., we can normalize the capacities by setting $b_i := 1$ for all $i \in V$ and $h(vw) := h(vw)/b_w$, $t(vw) := t(vw)/b_v$ for all $vw \in E$. Such normalization is known to be preferable to make the subgradient method stable, and we adopted this technique.

3.3. Column generation method

We employ the column generation method to generate candidate in-trees. It starts from an initial in-tree set $T \subseteq T_{\text{all}}$ and repeatedly augments T until a stopping criterion is satisfied.

Let $T^+(\lambda)$ be the set of all in-trees having positive relative costs $c_j(\lambda) > 0$ for a Lagrangian multiplier vector λ (i.e., $T^+(\lambda) = \{j \in T_{\text{all}} \mid c_j(\lambda) > 0\}$). It is clear from the method of solving $LR(T, \lambda)$ (see Section 2) that if a set of in-trees $T \subseteq T_{\text{all}}$ satisfies $T^+(\lambda) \subseteq T$, then an optimal solution to $LR(T, \lambda)$ is also optimal to $LR(T_{\text{all}}, \lambda)$. On the other hand, if there is an in-tree $\tau \in T_{\text{all}}$ which is not included in T and has a positive relative cost $c_\tau(\lambda) > 0$, then an optimal solution $x(\lambda)$ to $LR(T, \lambda)$ cannot be optimal for $LR(T_{\text{all}}, \lambda)$. It is therefore necessary to find a new in-tree $\tau \in T_{\text{all}} \setminus T$ that satisfies

$$\sum_{i \in V} a_{i\tau} \lambda_i < 1. \tag{3.2}$$

The problem of finding such an in-tree (column) is generally called the *pricing problem.*

Copyright © by ORSJ. Unauthorized reproduction of this article is prohibited.

We showed in [18] that this pricing problem can be efficiently solved if λ is a feasible solution to the dual of the LP relaxation problem of $P(T)$. To solve the pricing problem, the algorithm in our previous paper solves the problem of finding a new in-tree $\tau \in T_{\text{all}} \setminus T$ that satisfies

$$\sum_{i \in V} a_{i\tau} \lambda_i = \min_{j \in T_{\text{all}} \setminus T} \left(\sum_{i \in V} a_{ij} \lambda_i \right). \tag{3.3}$$

A nice feature of a dual feasible solution λ is that $c_j(\lambda) = 1 - \sum_{i \in V} a_{ij}\lambda_i \leq 0$ holds for all $j \in T$, and hence if an in-tree $\tau \in T_{\text{all}}$ satisfying (3.2) is found, then we can conclude that τ is new, i.e., $\tau \notin T$. Then the problem of finding a new in-tree τ that satisfies (3.3) is equivalent to the problem of finding an in-tree τ that minimizes the left-hand side of (3.2) among all in-trees in T_{all}. This problem is equivalent to the minimum weight rooted arborescence problem as shown in [18].

This 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 the minimum total edge cost. The problem can be solved in $O(|E||V|)$ time by Edmonds' algorithm [5]. Bock [2] and Chu and Liu [4] obtained similar results. Gabow et al. [8] presented the best results so far with an algorithm of time complexity $O(|E| + |V| \log |V|)$, which uses Fibonacci heap. We employed Edmonds' algorithm to solve this problem from the easiness of implementation.

When the pricing problem is solved for a Lagrangian multiplier vector, the nice feature of dual feasible solutions is not always satisfied, and the column generation method may not work; it may generate in-trees that are already in T. However, we observed through preliminary experiments that such duplicate generation is not frequent if good Lagrangian multiplier vectors are used. Based on this observation, we use Lagrangian multiplier vectors obtained by SubOpt.

To have higher probability of generating an in-tree not in T, our algorithm solves the pricing problem for more than one Lagrangian multiplier vector, and for this reason, we let the procedure SubOpt output ρ Lagrangian multiplier vectors that attain the best ρ upper bounds. Our column generation method solves the pricing problem for a Lagrangian multiplier vector $\lambda^{(k)}$ in the ascending order of k starting from $k = 1$ until a new in-tree $\tau \notin T$ is found or all $\lambda^{(1)}, \ldots, \lambda^{(\rho)}$ are checked. If a new in-tree is found, then it is added into the current set of in-trees T. On the other hand, if no new in-trees are found even after applying the column generation method to the ρ Lagrangian multiplier vectors, the entire procedure of generating in-trees stops.

3.4. Stopping criteria of the column generation method

In this subsection, we consider the stopping criteria of the column generation method. We introduce two stopping criteria and stop the algorithm when one of these criteria is satisfied.

The first one uses upper bounds of $\text{OPT}_{P(T_{\text{all}})}$. In our previous paper [18], we proposed a method that calculates an upper bound of $\text{OPT}_{P(T_{\text{all}})}$ from a given set of in-trees T and a nonnegative vector $\lambda \geq 0$. More precisely, this method creates a dual feasible solution of the LP relaxation problem of $P(T_{\text{all}})$. We observed through computational experiments that the method gives a tight upper bound if a good in-tree set T and an appropriate vector λ are given. We use this property as a stopping criterion of the algorithm. For the candidates of λ, we employed Lagrangian multiplier vectors obtained by SubOpt, and upper bounds of $P(T_{\text{all}})$ are calculated in each iteration of the column generation method. Let UB^* be the best upper bound found by then during the iteration of our column generation algorithm.

Copyright © by ORSJ. Unauthorized reproduction of this article is prohibited.

If T is not yet a good set of in-trees, UB^* is often updated in the following iterations. On the other hand, when T becomes a good set of in-trees (i.e, it includes most of valuable in-trees), UB^* is updated infrequently. Hence we stop the algorithm if UB^* is not updated in $|V|$ consecutive iterations.

The second stopping criterion is based on the overlapping of generated in-trees. When no new in-trees are found even after applying the column generation method to all ρ Lagrangian multiplier vectors obtained by SubOpt, we stop the algorithm (as stated in Section 3.3).

In the computational experiments in Section 4, we set the value of parameter ρ to 10. The value of parameter ρ has little influence on the performance of the algorithm as long as it is sufficiently large. Indeed, this value $\rho = 10$ was large enough in our experiments because with this value of ρ, the proposed algorithm never stopped with the second stopping criterion.

3.5. Proposed algorithm to generate in-trees

The new algorithm to generate in-trees based on the column generation approach with the Lagrangian relaxation is formally described as Algorithm LRGenInTrees.

Algorithm LRGenInTrees

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 ρ.

Output: a set of in-trees T.

1: Create the initial set T_0 of $|V|$ in-trees randomly. Set $T := T_0$, $\mathrm{UB}^* := +\infty$, $\ell := 0$, $\lambda_i := 1/\min_{j \in T} \sum_{v \in V} a_{vj}$ for all $i \in V$ and $\pi := 2$.
2: Invoke PackInTrees and let LB be the obtained lower bound of $P(T)$.
3: Invoke $\textsc{SubOpt}(T, \mathrm{LB}, \lambda, \pi)$ to obtain $\lambda^{(1)}, \ldots, \lambda^{(\rho)}$ and $\pi^{(1)}, \ldots, \pi^{(\rho)}$, and set $\ell := \ell + 1$.
4: **for** $k = 1$ **to** ρ **do**
5: Calculate an upper bound UB of $\mathrm{OPT}_{P(T_{\mathrm{all}})}$ using the current in-tree set T and a vector $\lambda^{(k)}$ (by the method described in Section 3.4), and let $\mathrm{UB}^* := \mathrm{UB}$ and $\ell := 0$ if $\mathrm{UB} < \mathrm{UB}^*$.
6: Solve the pricing problem for a vector $\lambda^{(k)}$ and let τ be the generated in-tree.
7: If $\tau \notin T$ holds, then set $T := T \cap \{\tau\}$, $\lambda := \lambda^{(k)}$ and $\pi := 4\pi^{(k)}$, and go to Line 10.
8: **end for**
9: Output the set of in-trees T and stop.
10: If $\ell = |V|$ holds, then go to Line 9.
11: If a hundred new in-trees are added into T after the last call to PackInTrees, then invoke PackInTrees and update LB.
12: Return to Line 3.

3.6. Speed-up techniques

We propose two speed-up techniques for the subgradient method. The first is to decrease the number of in-trees used by the subgradient method. We observed through preliminary experiments that in-trees generated during an early period of the algorithm were not useful in executing the subgradient method. For example, even when about $|V|$ in-trees are added into the initial in-tree set, almost all in-trees in the initial in-tree set are never used in any optimal solution of $LR(T, \lambda)$ during the execution of SubOpt (i.e., $x_j(\lambda) = 0$ for almost all in-trees $j \in T_0$ in all iterations of SubOpt). Note that if $x_j(\lambda) = 0$ holds for all multipliers λ generated during the execution of SubOpt, the removal of the in-tree j does not affect the behavior of SubOpt. Based on this observation, we incorporate a mechanism to remove such “unnecessary” in-trees. Because it is not possible to detect unnecessary in-trees before executing SubOpt, we adopt a simple estimate based on the search history:

Copyright © by ORSJ. Unauthorized reproduction of this article is prohibited.

The algorithm removes in-trees that have been used very rarely during the recent calls to SubOpt. More precisely, LRGenInTrees invokes SubOpt(T', LB, λ, π) instead of SubOpt(T, LB, λ, π), where T' is the in-tree set obtained by removing unnecessary in-trees from T through the following rule. Whenever SubOpt is invoked, after its execution is terminated, the algorithm marks every in-tree whose number of times used as optimal solutions of $LR(T, \lambda)$ during this invocation of SubOpt is less than α, where α is a parameter for adjusting the number of in-trees to be judged unnecessary. We define T' as the in-tree set obtained from T by removing all in-trees that are marked in β (a parameter) successive calls to SubOpt during the execution of LRGenInTrees. We set $\alpha = 5$ and $\beta = |V|$ in our computational experiments.

The second is to reduce the practical computation time for each iteration of the subgradient method. One iteration of SubOpt is shown from Line 3 to Line 10 of algorithm SubOpt. If we implement SubOpt in a straightforward manner, Lines 3 and 7 take $\Theta(|V||T|)$ time, which are the bottlenecks, and other lines take $O(|V|)$ time. Below we explain the ideas to speed up Lines 3 and 7 of SubOpt.

The optimal value $\mathrm{OPT}_{LR(T,\lambda)}$ and an optimal solution $x(\lambda)$ of $LR(T, \lambda)$ are calculated in Line 3. First, we discuss the method to compute an optimal solution $x(\lambda)$. Recalling the equation (2.4), to calculate an optimal solution $x(\lambda)$ from scratch when the Lagrangian multiplier vector λ is updated, relative costs $c_j(\lambda)$ for all $j \in T$ must be determined. Because the computation of $c_j(\lambda)$ takes $O(|V|)$ time for each $j \in T$, the calculation of an optimal solution $x(\lambda)$ takes $\Theta(|V||T|)$ time if naively implemented. However, we note that if $x(\lambda')$ is available for the multiplier λ' of the previous iteration, it is only necessary to update $x_j(\lambda)$ for those in-trees j whose relative cost changes from $c_j(\lambda') > 0$ to $c_j(\lambda) \leq 0$, or vice versa. Moreover, because the change in the values of λ_i is small in every iteration (except for the early stage of SubOpt), the value of $x_j(\lambda)$ tends to stay the same (i.e., $x_j(\lambda) = x_j(\lambda')$) for most of the in-trees j. Based on this observation, we introduce upper and lower bounds of relative costs $c_j(\lambda)$, and the algorithm skips the computation of the exact value $c_j(\lambda)$ for every in-tree j such that the value of $x_j(\lambda)$ is easily determined from the upper or lower bound. Assume that we have an upper bound $c_j^{\mathrm{UB}}(\lambda)$ and a lower bound $c_j^{\mathrm{LB}}(\lambda)$ of relative cost $c_j(\lambda)$ for all in-trees $j \in T$. We can decide $x_j(\lambda) = 0$ when $c_j^{\mathrm{UB}}(\lambda) \leq 0$ and $x_j(\lambda) = u_j$ when $c_j^{\mathrm{LB}}(\lambda) > 0$ for each in-tree $j \in T$ even if we do not have the exact value of $c_j(\lambda)$. The algorithm then calculates the exact value of $c_j(\lambda)$ only if neither is satisfied (i.e., $c_j^{\mathrm{UB}}(\lambda) > 0$ and $c_j^{\mathrm{LB}}(\lambda) \leq 0$). If good upper bounds $c_j^{\mathrm{UB}}(\lambda)$ and good lower bounds $c_j^{\mathrm{LB}}(\lambda)$ are given, we can reduce the actual computation time, though the worst case time complexity does not change from $O(|V||T|)$. Before explaining the method we adopted to compute upper and lower bounds, we confirm that the omission of computing $c_j(\lambda)$ does not affect the computation of other parts of SubOpt.

Let us discuss the method of computing the optimal value $\mathrm{OPT}_{LR(T,\lambda)}$. Because we omit the calculation of the exact values of $c_j(\lambda)$ for some in-trees when obtaining an optimal solution $x(\lambda)$, it is no longer possible to calculate $\mathrm{OPT}_{LR(T,\lambda)}$ by the equation (2.3). Instead, we calculate $\mathrm{OPT}_{LR(T,\lambda)}$ by the equation (2.2). The equation (2.2) can be rewritten as follows:

$$\sum_{j\in T} x_j(\lambda) + \sum_{i\in V} \lambda_i \left(b_i - \sum_{j\in T} a_{ij} x_j(\lambda) \right) = \sum_{j\in T} x_j(\lambda) + \sum_{i\in V} \lambda_i s_i(\lambda).$$

If the subgradient $s(\lambda)$ is given, we can calculate $\mathrm{OPT}_{LR(T,\lambda)}$ in $O(|V| + |T|)$ time by this equation. In the pseudocode of Algorithm SubOpt, the subgradient $s(\lambda)$ is computed

Copyright © by ORSJ. Unauthorized reproduction of this article is prohibited.

after calculating the optimal value $\mathrm{OPT}_{LR(T,\lambda)}$. However, it is easy to see that there is no problem in computing the subgradient $s(\lambda)$ before obtaining $\mathrm{OPT}_{LR(T,\lambda)}$, i.e., the order of computation in Lines 3–7 is modified as follows: The optimal solution $x(\lambda)$ is first computed, and the subgradient $s(\lambda)$ is obtained next. The optimal value $\mathrm{OPT}_{LR(T,\lambda)}$ is then calculated, and the set Λ and the best upper bound $\mathrm{UB}_{\mathrm{best}}$ are updated if necessary.

We next focus on Line 7 in which the subgradient $s(\lambda)$ is calculated. As in the case of Line 3, it takes $\Theta(|V||T|)$ time if naively implemented. The equation (2.4) indicates that $x_j(\lambda)$ takes either 0 or u_j for all $j \in T$, and only those in-trees j with $x_j(\lambda) = u_j$ contribute to the calculation of the subgradient $s(\lambda)$, i.e., we can calculate $s(\lambda)$ as follows:

$$s_i(\lambda) = b_i - \sum_{\substack{j \in T \\ x_j(\lambda)=u_j}} a_{ij} x_j(\lambda).$$

In our preliminary experiments, we observed that most of the variables in an optimal solution have $x_j(\lambda) = 0$ for many iterations of SubOpt. We can therefore reduce the actual computation time though the worst case time complexity does not change from $O(|V||T|)$.

We then explain how our algorithm computes an upper bound $c_j^{\mathrm{UB}}(\lambda)$ and a lower bound $c_j^{\mathrm{LB}}(\lambda)$. In the first iteration of SubOpt, it calculates the exact value of relative cost $c_j(\lambda)$ and sets $c_j^{\mathrm{UB}}(\lambda) := c_j(\lambda)$ and $c_j^{\mathrm{LB}}(\lambda) := c_j(\lambda)$ for all in-trees $j \in T$. In the subsequent iterations, the algorithm updates $c_j^{\mathrm{UB}}(\lambda)$ and $c_j^{\mathrm{LB}}(\lambda)$ as follows. For the consumptions a_{ij} of in-tree j at node i defined by the equation (1.1), $a_i^{\max}$ and $a_i^{\min}$ are defined as

$$a_i^{\max} = \max_{j \in T} a_{ij}, \tag{3.4}$$

$$a_i^{\min} = \min_{j \in T} a_{ij}. \tag{3.5}$$

Assume that we have the Lagrangian multiplier vector λ' of the previous iteration, and consider the moment when λ' is updated to λ by the equation (3.1). We define $\Delta\lambda$ as the difference between the values of the Lagrangian multiplier vectors λ' and λ, i.e., $\lambda = \lambda' + \Delta\lambda$. The relative cost $c_j(\lambda)$ of the updated Lagrangian multiplier vector λ is as follows:

$$\begin{aligned} c_j(\lambda) = c_j(\lambda' + \Delta\lambda) &= 1 - \sum_{i \in V} a_{ij}(\lambda'_i + \Delta\lambda_i) \\ &= c_j(\lambda') - \sum_{i \in V} a_{ij} \Delta\lambda_i. \end{aligned}$$

By equations (3.4) and (3.5), if $c_j^{\mathrm{UB}}(\lambda') \le c_j(\lambda') \le c_j^{\mathrm{LB}}(\lambda')$ is satisfied, the following inequalities hold:

$$\begin{aligned} c_j(\lambda) = c_j(\lambda' + \Delta\lambda) &\le c_j^{\mathrm{UB}}(\lambda') - \sum_{\substack{i \in V \\ \Delta\lambda_i < 0}} a_i^{\max} \Delta\lambda_i - \sum_{\substack{i \in V \\ \Delta\lambda_i \ge 0}} a_i^{\min} \Delta\lambda_i, \\ c_j(\lambda) = c_j(\lambda' + \Delta\lambda) &\ge c_j^{\mathrm{LB}}(\lambda') - \sum_{\substack{i \in V \\ \Delta\lambda_i \ge 0}} a_i^{\max} \Delta\lambda_i - \sum_{\substack{i \in V \\ \Delta\lambda_i < 0}} a_i^{\min} \Delta\lambda_i. \end{aligned}$$

Using these results, we can obtain an upper bound $c_j^{\mathrm{UB}}(\lambda)$ and a lower bound $c_j^{\mathrm{LB}}(\lambda)$ of the

Copyright © by ORSJ. Unauthorized reproduction of this article is prohibited.

relative cost $c_j(\lambda)$ as follows:

$$c_j^{\mathrm{UB}}(\lambda) := c_j^{\mathrm{UB}}(\lambda') - \sum_{\substack{i\in V\\ \Delta\lambda_i<0}} a_i^{\max}\Delta\lambda_i - \sum_{\substack{i\in V\\ \Delta\lambda_i\geq 0}} a_i^{\min}\Delta\lambda_i, \tag{3.6}$$

$$c_j^{\mathrm{LB}}(\lambda) := c_j^{\mathrm{LB}}(\lambda') - \sum_{\substack{i\in V\\ \Delta\lambda_i\geq 0}} a_i^{\max}\Delta\lambda_i - \sum_{\substack{i\in V\\ \Delta\lambda_i<0}} a_i^{\min}\Delta\lambda_i. \tag{3.7}$$

By using (3.6) and (3.7), we have upper and lower bounds $c_j^{\mathrm{UB}}(\lambda)$ and $c_j^{\mathrm{LB}}(\lambda)$ for each iteration of SubOpt except for the first iteration even if the computation of $c_j(\lambda)$ is omitted. Note that whenever the exact value of $c_j(\lambda)$ is computed, the values of $c_j^{\mathrm{UB}}(\lambda)$ and $c_j^{\mathrm{LB}}(\lambda)$ are set to $c_j(\lambda)$. The computation time of the update by equations (3.6) and (3.7) for all $j \in T$ is $O(|V| + |T|)$ and is small. The computation time of obtaining $a_i^{\max}$ and $a_i^{\min}$ for all $i \in V$ is $O(|V||T|)$, but we need to calculate $a_i^{\max}$ and $a_i^{\min}$ for all $i \in V$ only once at the beginning of SubOpt whenever it is invoked. This time complexity is the same as the time to compute the exact values of $c_j(\lambda)$ for all $j \in T$ in the first iteration of SubOpt and hence is sufficiently small, but we can further reduce the computation time of obtaining $a_i^{\max}$ and $a_i^{\min}$ for all $i \in V$ by using heaps, i.e., if we use two heaps to maintain $a_i^{\max}$ and $a_i^{\min}$ for all $i \in V$, which are updated whenever a new in-tree is added or an unnecessary in-tree is removed, the computation time throughout the execution of the proposed algorithm becomes $O(|V||T| \log |T|)$ time independent of the number of calls to SubOpt. We adopted this method in our implementation. Note that the calculation of $c_j^{\mathrm{UB}}(\lambda)$ and $c_j^{\mathrm{LB}}(\lambda)$ for all $j \in T$ never becomes the bottleneck in the iterations of SubOpt. The above rule to compute the upper bound $c^{\mathrm{UB}}(\lambda)$ and the lower bound $c^{\mathrm{LB}}(\lambda)$ is very simple, but we observed that this technique is quite effective in reducing the computation time of SubOpt.

According to our comparison between the basic algorithm and the one incorporated with all above speed-up techniques, the execution time per call to SubOpt became about 2–4 times faster.

3.7. Method to obtain feasible solutions

We proposed an algorithm to generate a set of in-trees in the previous sections. To evaluate the performance of the proposed algorithm on the node capacitated in-tree packing problem, a method to obtain a feasible solution of $P(T_{\mathrm{all}})$ is necessary. Based on the second-phase algorithm proposed in [18], we devise a heuristic method called PackInTrees*.

Let T_0 be the initial set of in-trees and T_k be the set of in-trees T after the kth iteration of LRGenInTrees for $k = 1, \ldots, f$, where f is the number of in-trees generated by LRGenInTrees. The procedure PackInTrees* solves the LP relaxation problems of $P(T_{f-\gamma}), \ldots, P(T_f)$ and obtains an optimal solution for each problem, where γ is a parameter that we set $\gamma = 10$ in our computational experiments. For each optimal solution x^* of the LP relaxation problems, a feasible solution of $P(T_{\mathrm{all}})$ is generated by rounding down every variable x_j^* of the solution, and then it is improved by applying PackInTrees, which is the greedy algorithm proposed in [18]. Among the γ feasible solutions obtained by this procedure, PackInTrees* outputs the best one.

4. Computational Experiments

4.1. Instances and experimental environment

We use two types of instances in our experiments. The first one is based on sensor location data used by Heinzelman et al. [10] and Sasaki et al. [17] in their papers about sensor

Copyright © by ORSJ. Unauthorized reproduction of this article is prohibited.

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. To be more precise, as in [10] and [17], we use parameters $E_{\text{elec}} = 50$ nJ/bit, $\varepsilon_{\text{fs}} = 10$ pJ/bit/m^2, $\varepsilon_{\text{mp}} = 0.0013$ pJ/bit/m^4, $E_{\text{DA}} = 5$ nJ/bit/signal, $l = 4200$ bit and $d_0 = 87$ m. Defining $d(vw), vw \in E$ as the Euclidean distance from a vertex v to a vertex w, we use the following consumption functions

$$t(vw) = \begin{cases} lE_{\text{elec}} + l\varepsilon_{\text{fs}}\{d(vw)\}^2, & \text{if } d(vw) < d_0, \\ lE_{\text{elec}} + l\varepsilon_{\text{mp}}\{d(vw)\}^4, & \text{if } d(vw) \geq d_0, \end{cases}$$

$$h(vw) = lE_{\text{elec}} + lE_{\text{DA}}$$

and capacities $b_i = 0.5$ J for all $i \in V$. We call the instances hcb100, sfis100-1, sfis100-2 and sfis100-3, where hcb100 is the instance generated using the sensor location data in [10], and sfis100-1, 2 and 3 are the instances generated using the sensor location data called data1, 2 and 3, respectively, in [17].

The second type consists of randomly generated instances. We named them "rnd$n$-δ-b-(h, t or none)," where n is the number of nodes, δ is the edge density, b is the capacity of all $i \in V^-$ (where $V^- = V \setminus \{r\}$) and h, t or none shows which of head and tail consumptions is bigger (i.e., "h" implies that head consumptions are bigger than tail consumptions, "t" implies that tail consumptions are bigger than head consumptions, and no sign implies head and tail consumptions are chosen from the same range). We generated instances with $n = 100, 200$, $\delta = 5\%, 50\%$ and $b = 10000, 100000$ ($+\infty$ for the root node r). Instances of $\delta = 5\%$ (50%) are generated so that the out-degree of each node ranges from 4% (40%) to 6% (60%) of the number of nodes. Tail and head consumptions for "h" instances were randomly chosen from the integers in the intervals $[3, 5]$ and $[30, 50]$, respectively, for all edges not connected to the root. Similarly, those for "t" instances were randomly chosen from $[30, 50]$ and $[3, 5]$, and those for instances without "h" or "t" sign were randomly chosen from $[30, 50]$ and $[30, 50]$. The tail consumption of edges entering the root node r for all instances were randomly chosen from the integers in the interval $[300, 500]$ so that these edges cannot be used frequently.

The algorithms were coded in the C++ language and ran on a Dell PowerEdge T300 (Xeon X3363 2.83GHz, 6MB cache, 24GB memory), where the computation was executed on a single core. We used the primal simplex method in GLPK4.43* as LP solver.

4.2. Experimental results

Figure 1 represents the behavior of the proposed algorithm LRGenInTrees and the previous algorithm GenInTrees applied to rnd200-50-100000-h. The horizontal and the vertical axes represent the number of in-trees generated by algorithms and the objective value, respectively. (Note that to draw this figure, Algorithm LRGenInTrees was not terminated with its standard stopping criterion even though the stopping criterion was satisfied before 3000 in-trees were generated.) The figure shows the improvement of the best upper bounds of the original problem $P(T_{\text{all}})$ and the upper bounds of the subproblem $P(T)$ as in-trees are added into T at each iteration. For LRGenInTrees, the upper bound of $P(T)$ means the best optimal value $\text{OPT}_{LR(T,\lambda)}$ for all λ generated by the latest call to SubOpt before T is updated, and for GenInTrees, the upper bound of $P(T)$ is the optimal value of the

*GLPK – GNU Project – Free Software Foundation (FSF), http://www.gnu.org/software/glpk/, 20 July 2011.

Copyright © by ORSJ. Unauthorized reproduction of this article is prohibited.

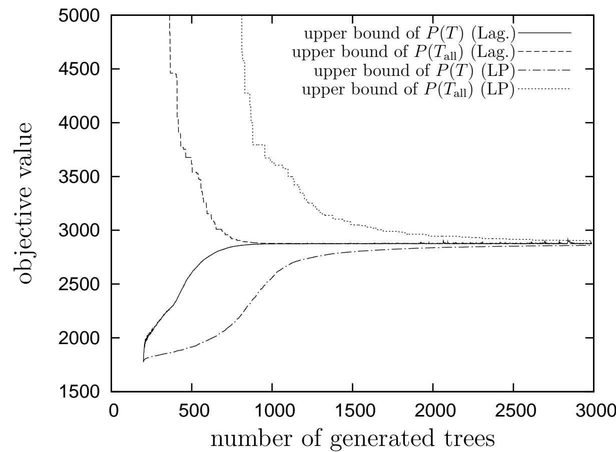

Figure 1: Behavior of the proposed and the previous first-phase algorithms LRGenInTrees and GenInTrees applied to rnd200-50-100000-h ("Lag." and "LP" represent LRGenInTrees and GenInTrees, respectively)

LP relaxation problem for T. Along with their improvement, the difference between two upper bounds becomes smaller and the ratio of improvement decreases. In general, this tendency is often observed when applying the column generation method. We can observe that two upper bounds of LRGenInTrees converge much faster than GenInTrees, i.e., LRGenInTrees generates in-trees necessary to obtain good upper bounds of $P(T_{\text{all}})$ much earlier than GenInTrees. Such a set of in-trees tends to be useful to obtain good feasible solutions of $P(T_{\text{all}})$. We also observed a similar behavior for the other instances.

Table 1 shows the results of the proposed algorithm for the problem instances explained in Section 4.1. It also shows the results of existing algorithms [17, 18] for comparison purposes. The first three columns represent instance names, the number of nodes $|V^-|$ (without the root node), and the number of edges $|E|$. Column $\text{UB}_{\text{b.k.}}$ shows the best-known upper bounds of $\text{OPT}_{P(T_{\text{all}})}$ computed by the algorithm in [18] allowing long computation time. The next columns include the experimental results of the proposed algorithm and the previous algorithm [18]. Column $|T|$ shows the number of in-trees generated by the algorithm LRGenInTrees, and column UB^* shows the best upper bound of $\text{OPT}_{P(T_{\text{all}})}$ obtained by LRGenInTrees. The next three columns represent objective values, denoted "Obj.," the gaps in % between $\text{UB}_{\text{b.k.}}$ and Obj., i.e., $((\text{UB}_{\text{b.k.}} - \text{Obj.})/\text{UB}_{\text{b.k.}}) \times 100$, and computation times in seconds. The last column SFIS shows the results obtained by our implementation of the algorithm in [17]. Because their program code was specialized to Euclidean instances and was not applicable to randomly generated instances, we implemented their algorithm to apply it to our instances. To make the comparison fair, the previous algorithm [18] was stopped when it generated the same number of in-trees as the new algorithm, and we set the time limit of SFIS to 10 seconds for instances with $|V^-| = 100$ and 100 seconds for instances with $|V^-| = 200$.

The results presented in Table 1 show that the proposed algorithm obtains better results than the previous algorithm and SFIS. The proposed algorithm attains better objective values than the previous algorithm even though its computation time is shorter (except for some instances with $|V^-| = 100$) and the number of generated in-trees is the same. The computation time of the proposed algorithm is about one minute on average for instances with $|V^-| = 200$, and the number of in-trees generated by the proposed algorithm is about $7|V^-|$ on average. The gaps between upper bounds and objective values are quite small;

Copyright © by ORSJ. Unauthorized reproduction of this article is prohibited.

Table 1: Computational results of three algorithms

Instance name	$\|V^-\|$	$\|E\|$	$UB_{b.k.}$	Proposed Algorithm					Prev. Algorithm [18]			SFIS
				$\|T\|$	UB^*	Obj.	Gap	Time	Obj.	Gap	Time	
hcb100	100	10100	1124	684	1125	1119	0.44	6.2	1092	2.85	6.0	930
sfis100-1	100	10100	1097	695	1098	1089	0.73	8.4	1081	1.46	6.1	1032
sfis100-2	100	10100	1097	556	1098	1090	0.64	5.8	1059	3.46	3.9	1032
sfis100-3	100	10100	1101	696	1102	1095	0.54	8.8	1089	1.09	6.6	1021
rnd100-5-10000-h	100	473	225	627	225	216	4.00	4.1	181	19.56	5.7	159
rnd100-5-10000-t	100	473	217	788	229	217	0.00	7.1	191	11.98	7.5	209
rnd100-5-10000	100	473	128	608	128	123	3.91	3.8	108	15.63	5.4	99
rnd100-5-100000-h	100	473	2251	605	2252	2243	0.36	5.3	1874	16.75	5.2	1611
rnd100-5-100000-t	100	473	2173	869	2302	2173	0.00	12.0	2046	5.84	9.2	2108
rnd100-5-100000	100	473	1283	691	1283	1276	0.55	5.4	1160	9.59	7.1	1003
rnd100-50-10000-h	100	4938	272	931	272	263	3.31	10.4	261	4.04	10.9	211
rnd100-50-10000-t	100	4938	270	656	270	257	4.81	7.1	186	31.11	6.3	238
rnd100-50-10000	100	4938	149	542	149	143	4.03	4.2	132	11.41	3.8	119
rnd100-50-100000-h	100	4938	2726	784	2726	2717	0.33	9.9	2673	1.94	7.8	2292
rnd100-50-100000-t	100	4938	2701	628	2701	2688	0.48	9.2	1945	27.99	6.0	2413
rnd100-50-100000	100	4938	1498	597	1499	1490	0.53	5.1	1430	4.54	4.6	1295
rnd200-5-10000-h	200	1970	260	1065	260	247	5.00	20.4	185	28.85	70.9	172
rnd200-5-10000-t	200	1970	250	1323	261	249	0.40	39.1	183	26.80	104.0	229
rnd200-5-10000	200	1970	141	898	141	131	7.09	15.4	104	26.24	55.6	106
rnd200-5-100000-h	200	1970	2602	1451	2603	2583	0.73	56.7	2152	17.29	149.8	1755
rnd200-5-100000-t	200	1970	2500	1291	2622	2500	0.00	52.8	1943	22.28	101.0	2302
rnd200-5-100000	200	1970	1411	1005	1412	1401	0.71	22.7	1173	16.87	70.3	1091
rnd200-50-10000-h	200	20030	287	1652	287	273	4.88	73.3	265	7.67	153.6	200
rnd200-50-10000-t	200	20030	286	1329	286	273	4.55	58.9	179	37.41	109.0	250
rnd200-50-10000	200	20030	156	987	157	146	6.41	29.0	107	31.41	51.8	111
rnd200-50-100000-h	200	20030	2874	1525	2876	2857	0.59	76.5	2779	3.31	123.9	2374
rnd200-50-100000-t	200	20030	2867	1270	2868	2855	0.42	78.1	1920	33.03	100.9	2542
rnd200-50-100000	200	20030	1569	929	1570	1552	1.08	28.8	1299	17.21	48.2	1332

Copyright © by ORSJ. Unauthorized reproduction of this article is prohibited.

they are especially small for instances with $b = 100000$ and are less than 1% except for the last instance. Moreover, the proposed algorithm found exact optimal solutions for three instances.

Table 2 shows the computation time and the number of generated in-trees required by the previous algorithm to attain the solution value of the proposed algorithm. Columns in Table 2 have the same meaning as columns in Table 1 except for column Time, and the first five columns are taken from Table 1. Column Time shows the computation time spent for the first phase of generating in-trees, i.e., it does not include the time spent for the greedy algorithm of the second phase (the reason for reporting such computation time is explained in the remark below). The next three columns show the results of the previous algorithm when it attained the solution value of the proposed algorithm (except for those it failed in obtaining such a solution). For the instances with a symbol "$*$" in column Obj., the previous algorithm was not able to attain the solution value of the proposed algorithm before its stopping criterion was satisfied. For such instances, the table shows the results of the previous algorithm when it stopped with its original stopping criterion. The results indicate that to obtain a solution value similar to the proposed algorithm, the previous algorithm needs to generate a larger number of in-trees and this takes a long computation time. This tendency is clearer for instances with $|V^-| = 200$.

Remark. To take the data in Table 2, we needed to modify the previous algorithm for the following reason. The previous algorithm generates a set of in-trees T in the first phase and then applies the greedy method PackInTrees to the whole T starting from a small number of good feasible solutions. That is, no feasible solutions are generated in the intermediate stage of the first phase. Our objective here, however, is to observe how the quality of the in-tree set improves with the number of iterations of the previous first-phase algorithm GenInTrees. For this purpose, we slightly modified the previous algorithm so that it generates a feasible solution whenever a new in-tree is added. For this reason, the results of the previous algorithm in Table 2 are not necessarily the same as those reported in [18] even when the algorithm stopped with its original stopping criterion. Because the modified version of the previous algorithm applies the greedy method PackInTrees to more feasible solutions than the original one, it spends much longer computation time than the original. The proposed algorithm, on the other hand, applies the greedy method to a limited number of feasible solutions as in the case of the original version of the previous algorithm, and hence it spends much less computation time for the second phase than the modified version of the previous algorithm. We thus reported the computation time without the execution time of the greedy method to make the comparison fair.

5. Conclusions

In this paper, we proposed an algorithm to generate promising candidate in-trees for the node capacitated in-tree packing problem. This new algorithm generates a set of in-trees employing the subgradient method and the column generation method for the Lagrangian relaxation of the problem. We incorporated various ideas to speed up the algorithm, e.g., rules to decrease the number of in-trees used by the subgradient method, and to reduce the practical computation time for each iteration of the subgradient method.

The proposed algorithm obtained solutions whose gaps to the upper bounds are quite small, and was proved to be more efficient than existing algorithms.

Copyright © by ORSJ. Unauthorized reproduction of this article is prohibited.

Table 2: The computation time and the number of generated in-trees required by the previous algorithm to attain the solution value of the proposed algorithm

Instance name	$\|V^-\|$	$\|E\|$	Proposed Algorithm			Prev. Algorithm [18]		
			$\|T\|$	Obj.	Time†	$\|T\|$	Obj.	Time†
hcb100	100	10100	684	1119	5.9	1501	1119	27.4
sfis100-1	100	10100	695	1089	8.0	944	1089	10.8
sfis100-2	100	10100	556	1090	5.5	952	1090	11.1
sfis100-3	100	10100	696	1095	8.3	1036	1095	13.8
rnd100-5-10000-h	100	473	627	216	3.7	1492	216	23.8
rnd100-5-10000-t	100	473	788	217	7.0	1227	206*	14.7
rnd100-5-10000	100	473	608	123	3.4	1520	123	25.2
rnd100-5-100000-h	100	473	605	2243	4.9	2007	2243	40.3
rnd100-5-100000-t	100	473	869	2173	11.9	1227	2162*	14.7
rnd100-5-100000	100	473	691	1276	4.9	1963	1276	39.1
rnd100-50-10000-h	100	4938	931	263	9.7	1120	263	14.7
rnd100-50-10000-t	100	4938	656	257	6.6	1688	256*	23.7
rnd100-50-10000	100	4938	542	143	3.9	1096	143	14.2
rnd100-50-100000-h	100	4938	784	2717	9.5	1916	2717	40.8
rnd100-50-100000-t	100	4938	628	2688	8.7	1705	2688	24.4
rnd100-50-100000	100	4938	597	1490	4.7	1645	1490	30.6
rnd200-5-10000-h	200	1970	1065	247	17.5	4408	247	1474.9
rnd200-5-10000-t	200	1970	1323	249	38.3	5422	237*	1584.1
rnd200-5-10000	200	1970	898	131	13.2	3591	131	984.2
rnd200-5-100000-h	200	1970	1451	2583	52.3	6799	2583	3547.5
rnd200-5-100000-t	200	1970	1291	2500	52.6	5422	2485*	1583.3
rnd200-5-100000	200	1970	1005	1401	20.0	7224	1401	4105.4
rnd200-50-10000-h	200	20030	1652	273	68.3	2563	273	441.1
rnd200-50-10000-t	200	20030	1329	273	54.6	6062	273	1912.2
rnd200-50-10000	200	20030	987	146	26.7	2522	146	431.0
rnd200-50-100000-h	200	20030	1525	2857	71.9	4448	2857	1563.1
rnd200-50-100000-t	200	20030	1270	2855	73.7	7438	2855	2752.1
rnd200-50-100000	200	20030	929	1552	26.8	3699	1552	1073.5

† The computation time in these columns is the time spent for the first phase of generating in-trees, i.e., it does not include the time spent for the greedy algorithm of the second phase.

Copyright © by ORSJ. Unauthorized reproduction of this article is prohibited.

References

[1] E. Balas and A. Ho: Set covering algorithms using cutting planes, heuristics, and subgradient optimization: A computational study. *Mathematical Programming Study*, **12** (1980), 37–60.

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

[3] G. Calinescu, S. Kapoor, A. Olshevsky, and A. Zelikovsky: Network lifetime and power assignment in ad-hoc wireless networks. In G.D. Battista and U. Zwick (eds.): *Proceedings of the 11th European Symposium on Algorithms*, *Lecture Notes in Computer Science*, **2832** (Springer, 2003), 114–126.

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

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

[6] J. Edmonds: Edge-disjoint branchings. In B. Rustin (ed.): *Combinatorial Algorithms* (Academic Press, 1973), 91–96.

[7] M.L. Fisher: The Lagrangian relaxation method for solving integer programming problems. *Management Science*, **27** (1981), 1–18.

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

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

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

[11] M. Held and R.M. Karp: The traveling salesman problem and minimum spanning trees: Part II. *Mathematical Programming*, **1** (1971), 6–25.

[12] S. Imahori, Y. Miyamoto, H. Hashimoto, Y. Kobayashi, M. Sasaki, and M. Yagiura: The complexity of the node capacitated in-tree packing problem. *Networks*, to appear.

[13] N. Kamiyama, N. Katoh, and A. Takizawa: Arc-disjoint in-trees in directed graphs. *Combinatorica*, **29** (2009), 197–214.

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

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

[16] P.A. Pevzner: Branching packing in weighted graphs. *American Mathematical Society Translations*, **158** (2) (1994), 185–200.

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

[18] Y. Tanaka, S. Imahori, M. Sasaki, and M. Yagiura: An LP-based heuristic algorithm for the node capacitated in-tree packing problem. *Computers & Operations Research*, **39** (2012), 637–646.

Copyright © by ORSJ. Unauthorized reproduction of this article is prohibited.

Yuma Tanaka
Department of Computer Science and Mathematical Informatics
Graduate School of Information Science
Nagoya University
Furo-cho, Chikusa-ku, Nagoya, 464-8603, Japan
E-mail: tanaka@al.cm.is.nagoya-u.ac.jp

Copyright © by ORSJ. Unauthorized reproduction of this article is prohibited.