Available online at www.sciencedirect.com
SCIENCE DIRECT®
European Journal of Operational Research 172 (2006) 472–499
EUROPEAN JOURNAL OF OPERATIONAL RESEARCH
www.elsevier.com/locate/ejor

Discrete Optimization

A 3-flip neighborhood local search for the set covering problem

Mutsunori Yagiura [a,*], Masahiro Kishida [b,1], Toshihide Ibaraki [c]

[a] Department of Applied Mathematics and Physics, Graduate School of Informatics, Kyoto University, Kyoto 606-8501, Japan
[b] Sumitomo Electric Information Systems Co., Ltd., 2-1-3, Nishimiyahara, Yodogawa-ku, Osaka 532-0004, Japan
[c] Department of Informatics, School of Science and Technology, Kwansei Gakuin University, 2-1 Gakuen, Sanda 669-1337, Japan

Received 16 May 2003; accepted 29 October 2004
Available online 1 January 2005

Abstract

The set covering problem (SCP) calls for a minimum cost family of subsets from n given subsets, which together covers the entire ground set. In this paper, we propose a local search algorithm for SCP, which has the following three characteristics. (1) The use of 3-flip neighborhood, which is the set of solutions obtainable from the current solution by exchanging at most three subsets. As the size of 3-flip neighborhood is $O(n^3)$, the neighborhood search becomes expensive if implemented naively. To overcome this, we propose an efficient implementation that reduces the number of candidates in the neighborhood without sacrificing the solution quality. (2) We allow the search to visit the infeasible region, and incorporate the strategic oscillation technique realized by adaptive control of penalty weights. (3) The size reduction of the problem by using the information from the Lagrangian relaxation is incorporated, which is indispensable for solving very large instances. According to computational comparisons on benchmark instances with other existing heuristic algorithms for SCP, our algorithm performs quite effectively for various types of problems, especially for very large-scale instances.

Keywords: Metaheuristics; Set covering problem; Large neighborhood; Strategic oscillation; Lagrangian relaxation

1. Introduction

1.1. Problem definition and historical background

The set covering problem (SCP) is one of the representative combinatorial optimization problems. Given the ground set of m elements $i \in M = \{1, \ldots, m\}$, n subsets $S_j \subseteq M$, $j \in N = \{1, \ldots, n\}$, and costs c_j (>0)

* Corresponding author. Tel.: +81 75 753 5494; fax: +81 75 761 2437.
E-mail addresses: yagiura@i.kyoto-u.ac.jp (M. Yagiura), kishida-masahiro@sei.co.jp (M. Kishida), ibaraki@ksc.kwansei.ac.jp (T. Ibaraki).
[1] Tel.: +81 6 6394 6796; fax: +81 6 6394 6762.

0377-2217/$ - see front matter
doi:10.1016/j.ejor.2004.10.018

for subsets S_j, we want to choose a family of subsets S_j with the minimum total cost under the constraint that each element $i \in M$ is covered by at least one subset in the family. The SCP can be described as follows:

$$\text{SCP}: \text{minimize } cost(\boldsymbol{x}) = \sum_{j \in N} c_j x_j$$

$$\text{subject to } \sum_{j \in N} a_{ij} x_j \geqslant 1 \quad \forall i \in M,$$

$$x_j \in \{0, 1\} \quad \forall j \in N,$$

where

$$a_{ij} = \begin{cases} 1 & (\text{if } i \in S_j), \\ 0 & (\text{otherwise}). \end{cases}$$

It is understood that variable x_j equals 1 if subset S_j is in the selected family and 0 otherwise.

The SCP has many practical applications, including bus, railway and airline crew scheduling [2,11,23], location problems of facilities [24], and logical analysis of data [8]. It is known to be NP-hard in the strong sense, and a number of exact and heuristic algorithms have been proposed in the literature.

Among exact branch-and-bound algorithms, Fisher and Kedia proposed an exact branch-and-bound algorithm based on a dual heuristic, and solved SCP instances with up to 200 rows and 2000 columns [15]. Beasley combined a Lagrangian-based heuristic, feasible solution exclusion constraints and Gomory *f*-cuts, and improved the branching strategy to enhance his previous algorithm [4]. This algorithm could solve instances with up to 400 rows and 4000 columns [6].

Approximate algorithms have also been extensively studied [1,5,7,9–11,17,21]. Brusco et al. [9,21] developed simulated annealing algorithms, Beasley and Chu [7] presented a genetic algorithm, and Ceria et al. [11] presented a Lagrangian-based heuristic for solving very large-scale instances. Caprara et al. [10] combined a Lagrangian-based heuristic and greedy algorithm, and obtained impressive results for various types of instances, especially for very large-scale instances with up to 5000 rows and 1,000,000 columns. All these algorithms can be thought of as based on local search, in which the 1-flip neighborhood (i.e., the set of solutions obtainable by adding a subset to or deleting a subset from the current family) is used.

1.2. General ideas for the proposed algorithm

In the local search, how to define the neighborhood is crucial. For a positive integer r, let the r-flip neighborhood of a solution $\boldsymbol{x} = (x_1, \ldots, x_n)$ be the set of solutions obtainable by flipping at most r variables of $\boldsymbol{x}$. To our knowledge, most of the previous heuristic algorithms for SCP use 1-flip neighborhood. In this paper, we propose a local search algorithm with 3-flip neighborhood. As the size of 3-flip neighborhood is $O(n^3)$, which is much larger than that of 1-flip neighborhood, the search in the 3-flip neighborhood becomes quite expensive if it is conducted without deliberate consideration. To overcome this, we propose an implementation that effectively reduces the number of candidates in the neighborhood without sacrificing the solution quality.

We also incorporate a strategic oscillation mechanism [16,19], to guide the search between feasible and infeasible regions alternately (i.e., to intensify the search around the boundary of feasible region). Call a feasible solution $\boldsymbol{x}$ *minimal* if it becomes infeasible by flipping any variable x_j with $x_j = 1$. An optimal solution for SCP is always minimal, and strategic oscillation may be understood as a tool to search minimal solutions effectively. The alternate use of greedy and stingy methods, used by Feo and Resende [13] and Jacobs and Brusco [21], is a simple form of the strategic oscillation. It consists of two phases, constructive

phase and destructive phase. In the constructive phase, a feasible solution is constructed by using the greedy method, and in the destructive phase the stingy method is used to obtain a minimal solution or an infeasible solution. In our algorithm, the strategic oscillation is realized in a more sophisticated manner. The search into the infeasible region is guided by an infeasibility measure evaluated by the penalized objective function, which is defined to be the sum of $cost(\boldsymbol{x})$ and the penalty weights p_i (> 0) for those $i \in M$ not covered by $\boldsymbol{x}$. The penalty weights p_i in the objective function are then adaptively controlled to intensify the search around the boundary between feasible and infeasible regions.

In addition, in order to handle very large-scale instances, we reduce the problem size by fixing some variables x_j to 0 or 1 on the basis of the information obtained from Lagrangian relaxation of SCP, where the set of fixed variables is heuristically adjusted during the algorithm. The information from Lagrangian relaxation is also used to control the search in the neighborhood. This type of heuristic size reduction is indispensable for tackling large instances as confirmed in [10,11].

1.3. Outline of the proposed algorithm

The proposed algorithm consists of two phases, the phase of local search and the phase of fixing variables. In both of these phases, the information from the Lagrangian relaxation problem is used, for which we need to solve the Lagrangian dual problem. For this purpose, we use the subgradient method. The relaxation and the subgradient method will be explained in Section 2.

The phase of local search is always applied to the instances whose sizes have been reduced in the phase of fixing variables. The search order in the neighborhood is controlled by using the information in the last call of the subgradient method of Lagrangian relaxation. As mentioned above, solutions are evaluated by the penalized objective function. Whenever the local search stops at a solution because the penalized objective value can not be improved, the penalty weights are updated and the local search resumes from the solution found in the previous local search. Thus the local search is executed many times, thereby realizing the strategic oscillation. The phase of local search is terminated if a sufficient number of local search iterations has been completed (the concrete rule for this is slightly complicated and will be explained in Section 5.2). The local search will be explained in Section 3, and how we update the penalty weights will be explained in Section 4.

The phase of fixing variables has two stages, the initial fixing stage and the modification stage. In the first stage, the subgradient method is applied to the Lagrangian dual of the given instance, and the information from the obtained solution is used to fix some variables x_j to 0. Then, whenever the phase of local search (applied to the reduced instance) stops, the set of fixed variables are heuristically adjusted by calling the modification stage of the phase of fixing variables. In the modification stage, a set N_1 of indices of variables x_j is chosen by analyzing the solution obtained in the previous phase of local search, and the subgradient method is applied to the Lagrangian dual of the instance in which variables x_j, $j \in N_1$, are fixed to 1. Then the information from the obtained solution to the Lagrangian dual is used to modify the set of variables x_j to be fixed to 0. The phase of fixing variables will be explained in Section 5.

The algorithm starts with the initial fixing stage, and then the phase of local search and the modification stage are alternately repeated until a global stopping criterion is satisfied.

1.4. Computational results

Finally we carried out computational experiment on various benchmark instances. The results show that our algorithm is quite effective, compared with other existing algorithms. Our algorithm could obtain the best solutions for almost all the tested instances. In particular, better solutions are found for four of the very large-scale instances [10] having up to 1,000,000 variables.

2. Lagrangian relaxation and the subgradient method

The information from the Lagrangian relaxation of SCP is used to reduce the problem size, and to control the search in the neighborhood. Though the contents in this section are well-known, we briefly summarize the basic ideas to keep the paper self-contained. In Section 2.1, we give the definition of Lagrangian relaxation problem, and in Section 2.2, we explain the subgradient method to find an approximate solution to the Lagrangian dual problem.

2.1. Lagrangian relaxation problem

The Lagrangian relaxation of SCP for a given Lagrangian multiplier vector $\boldsymbol{u} \in R_+^m$ (R_+ is the set of non-negative real numbers) is defined as follows:

$$\begin{aligned}\text{LR-SCP}(\boldsymbol{u}): \quad L(\boldsymbol{u}) &= \min_{\boldsymbol{x}\in\{0,1\}^n} \sum_{j\in N} c_j x_j + \sum_{i\in M} u_i \left(1 - \sum_{j\in N} a_{ij} x_j\right) \\ &= \min_{\boldsymbol{x}\in\{0,1\}^n} \sum_{j\in N} c_j(\boldsymbol{u}) x_j + \sum_{i\in M} u_i,\end{aligned}$$

where $c_j(\boldsymbol{u}) = c_j - \sum_{i\in M} a_{ij}u_i$ is the *relative cost* (or *reduced cost*) associated with j. For any $\boldsymbol{u} \in R_+^m$, $L(\boldsymbol{u})$ gives a lower bound on the optimal value of problem SCP; i.e., $L(\boldsymbol{u}) \leqslant cost(\boldsymbol{x})$ holds for any $\boldsymbol{u} \in R_+^m$ and any feasible solution $\boldsymbol{x}$. An optimal solution to LR-SCP($\boldsymbol{u}$), denoted by $\boldsymbol{x}(\boldsymbol{u})$, is obtained by setting $x_j(\boldsymbol{u}) := 1$ (respectively, 0) if $c_j(\boldsymbol{u}) < 0$ (respectively, if $c_j(\boldsymbol{u}) > 0$) and choosing the value of $x_j(\boldsymbol{u})$ from 0 or 1 arbitrarily if $c_j(\boldsymbol{u}) = 0$. In other words, $L(\boldsymbol{u})$ is given by $L(\boldsymbol{u}) = \sum_{j\in N} \min\{c_j(\boldsymbol{u}), 0\} + \sum_{i\in M} u_i$.

The Lagrangian dual problem LD-SCP asks to find a Lagrangian multiplier vector $\boldsymbol{u}^* \in R_+^m$ that maximizes $L(\boldsymbol{u})$ and is defined by

$$\text{LD-SCP}: \ \max\{L(\boldsymbol{u}) \mid \boldsymbol{u} \in R_+^m\}.$$

As problem LR-SCP($\boldsymbol{u}$) has the *integrality property* (i.e., its linear programming relaxation has an integer optimal solution), any optimal solution $\boldsymbol{u}^*$ to the dual of the LP relaxation of SCP

$$\begin{aligned}\text{LPD-SCP}: \text{maximize} \quad & \sum_{i\in M} u_i \\ \text{subject to} \quad & \sum_{i\in M} u_i a_{ij} \leqslant c_j \quad \forall j \in N, \\ & u_i \geqslant 0 \quad \forall i \in M,\end{aligned}$$

is also an optimal solution to the Lagrangian dual problem [14]. If a good Lagrangian multiplier vector $\boldsymbol{u}$ is obtained, the relative cost $c_j(\boldsymbol{u})$ gives a reliable information on the attractiveness of letting $x_j = 1$, because each j with $x_j = 1$ in an optimal solution of SCP tends to have a small $c_j(\boldsymbol{u})$ value.

2.2. The subgradient method

As discussed in Section 2.1, any optimal solution $\boldsymbol{u}^*$ to LPD-SCP is an optimal solution to LD-SCP; however, computing such $\boldsymbol{u}^*$ directly is usually quite expensive, especially for very large-scale instances. A common approach to compute an approximate $\boldsymbol{u}^*$ is the subgradient method [3,14,20]. It uses the subgradient vector $\boldsymbol{s}(\boldsymbol{u}) \in R^m$, associated with a given $\boldsymbol{u}$, defined by $s_i(\boldsymbol{u}) = 1 - \sum_{j\in N} a_{ij}x_j(\boldsymbol{u})$ for all $i \in M$. This method generates a sequence $\boldsymbol{u}^{(0)}, \boldsymbol{u}^{(1)}, \ldots$, where $\boldsymbol{u}^{(0)}$ is a given initial vector, and $\boldsymbol{u}^{(k+1)}$ is updated from $\boldsymbol{u}^{(k)}$ by the following formula:

Algorithm GREEDY

Step 1 Let $\boldsymbol{x} := 0$.

Step 2 Choose j that minimizes $\gamma_j(\boldsymbol{x}) = c_j \,/\, \left|\{i \in S_j \mid \sum_{j' \in N} a_{ij'} x_{j'} = 0\}\right|$ among the j's satisfying $\{i \in S_j \mid \sum_{j' \in N} a_{ij'} x_{j'} = 0\} \neq \emptyset$, and let $x_j := 1$.

Step 3 If $\{i \in M \mid \sum_{j \in N} a_{ij} x_j = 0\} = \emptyset$, output $\boldsymbol{x}$ and exit; otherwise return to Step 2.

Fig. 1. The greedy algorithm.

$$u_i^{(k+1)} := \max\left\{u_i^{(k)} + \lambda \frac{\mathrm{UB} - L(\boldsymbol{u}^{(k)})}{\|\boldsymbol{s}(\boldsymbol{u}^{(k)})\|^2} s_i(\boldsymbol{u}^{(k)}),\ 0\right\} \quad \forall i \in M,$$

where UB is an upper bound on $cost(\boldsymbol{x})$, and $\lambda \geqslant 0$ is a parameter, called the step size.

Various implementations of the subgradient method are possible; however, when large instances are solved, the computational time spent on this method becomes very large if a naive implementation is used. We therefore use the sophisticated implementation proposed in [10]. Let us denote the subgradient method using an upper bound UB and starting from the initial vector $\boldsymbol{u}^{(0)}$ as SUBGRADIENT(UB, $\boldsymbol{u}^{(0)}$), which returns the best Lagrangian multiplier vector found during its iteration. In our algorithm, procedure SUBGRADIENT(UB, $\boldsymbol{u}^{(0)}$) is called many times. In the first call, we set $u_i^{(0)} = \min\{c_j/|S_j| \mid i \in S_j\}$ $(\forall i \in M)$, and in other calls, we set $\boldsymbol{u}^{(0)} = \boldsymbol{u}^{+}$, where $\boldsymbol{u}^{+}$ is the Lagrangian multiplier vector obtained by the first call of SUBGRADIENT(UB, $\boldsymbol{u}^{(0)}$).

Though UB is usually set to the incumbent value (i.e., the cost of the best feasible solution obtained during the search so far), UB is unknown in the first call of SUBGRADIENT(UB, $\boldsymbol{u}^{(0)}$). Then we initialize UB by UB := $cost(\boldsymbol{x})$ for the feasible solution $\boldsymbol{x}$ obtained by the well known greedy method [2,3], which is described in Fig. 1.

3. Local search with the 3-flip neighborhood

In this section, we propose an efficient search method for the 3-flip neighborhood. The main idea is to reduce the number of candidates in the neighborhood. (For simplicity, we explain the method assuming that it is applied to the original instance of SCP, though it is usually applied to the reduced instances.)

3.1. The local search, search space and the neighborhood

The local search starts from an initial solution x and repeats replacing x with a better solution in its *neighborhood* NB(x) until no better solution is found in NB(x). A solution x is called *locally optimal*, if no better solution exists in NB(x).

In our algorithm, we allow the search to visit infeasible region. In this case, the objective function itself is not appropriate to evaluate the quality of solutions. Instead, we use the following penalized cost function. Let p_i (> 0) be a penalty weight for each $i \in M$. A solution $\boldsymbol{x}$ is evaluated by

$$pcost(\boldsymbol{x}) = \sum_{j \in N} c_j x_j + \sum_{i \in M} p_i \max\left\{1 - \sum_{j \in N} a_{ij} x_j, 0\right\}.$$

The penalty weights p_i are adaptively controlled, as will be explained in Section 4. We emphasize at this point that $pcost(\boldsymbol{x})$ and $cost(\boldsymbol{x})$ are both used in our algorithm.

Remark. One might think that using a good Lagrangian multiplier vector $\boldsymbol{u}^*$ as the penalty weight $\boldsymbol{p} = (p_1, \ldots, p_m)$ would work. As this idea is quite natural, we tried this first; however, we found that the Lagrangian multipliers tend to be too small for this purpose.

For a positive integer r, the r-flip neighborhood $\mathrm{NB}_r(\boldsymbol{x})$ is defined by

$$\mathrm{NB}_r(\boldsymbol{x}) = \{\boldsymbol{x}' \in \{0,1\}^n \mid d(\boldsymbol{x}, \boldsymbol{x}') \leqslant r\},$$

where

$$d(\boldsymbol{x}, \boldsymbol{x}') = |\{j \in N \mid x_j \neq x'_j\}|$$

is the Hamming distance between $\boldsymbol{x}$ and $\boldsymbol{x}'$. In other words, $\mathrm{NB}_r(\boldsymbol{x})$ is the set of solutions obtainable from $\boldsymbol{x}$ by flipping at most r variables. In our algorithm, the r is set to 3.

Since the region searched in one application of the local search is limited, the local search is usually applied many times. When a locally optimal solution is obtained, the standard strategy of our algorithm is to update penalty weights and to resume the local search from the obtained locally optimal solution.

3.2. Efficient implementation of the neighborhood search

Our procedure finds a better solution in the 3-flip neighborhood of a solution $\boldsymbol{x}$ or concludes that $\boldsymbol{x}$ is locally optimal. For convenience, let

$$n' = \sum_{j \in N} x_j,$$

$$t = \max\left\{ \sum_{i \in M} a_{ij} \,\middle|\, j \in N \right\},$$

$$l = \max\left\{ \sum_{j \in N} a_{ij} \,\middle|\, i \in M \right\},$$

where n' denotes the number of variables which are equal to 1, t denotes the maximum number of elements in one subset S_j, and l denotes the maximum number of subsets covering one element. They always satisfy $n' \leqslant n$, $t \leqslant m$ and $l \leqslant n$, and in most cases $n' \ll n$, $t \ll m$ and $l \ll n$ hold. For a vector $\boldsymbol{x} \in \{0,1\}^n$ and a subset $J \subseteq N$, define $\boldsymbol{x} \updownarrow J$ by

$$\boldsymbol{x} \updownarrow J = (x'_1, \ldots, x'_n) \Longleftrightarrow x'_j = \begin{cases} 1 - x_j & (\text{if } j \in J), \\ x_j & (\text{otherwise}), \end{cases}$$

which is the vector obtained from $\boldsymbol{x}$ by flipping the variables in J. We denote the increase in $pcost(\boldsymbol{x})$ by

$$\Delta pcost(\boldsymbol{x}, J) = pcost(\boldsymbol{x} \updownarrow J) - pcost(\boldsymbol{x}).$$

We also define

$$\theta_i^{(\kappa)}(\boldsymbol{x}) = \begin{cases} 1 & (\text{if } \sum_{j \in N} a_{ij} x_j = \kappa), \\ 0 & (\text{otherwise}) \end{cases}$$

for $i \in M$ and $\kappa = 0,1,\ldots,l$, which is the indicator function representing whether element i is covered by κ subsets in the solution $\boldsymbol{x}$.

In order to improve efficiency, we search the solutions in $\mathrm{NB}_r(\boldsymbol{x}) \backslash \mathrm{NB}_{r-1}(\boldsymbol{x})$ for $r = 2$ and 3, in this order, only if $\boldsymbol{x}$ is locally optimal with respect to $\mathrm{NB}_{r-1}(\boldsymbol{x})$. Let *one-round* be the computation needed to find an improved solution in the neighborhood or to conclude that the current solution is locally optimal. If

implemented naively, the local search with $\mathrm{NB}_r(\boldsymbol{x})$ requires $\mathrm{O}(n^r t)$ one-round time for $r \geqslant 1$. In our algorithm to be described below, one-round time is reduced to $\mathrm{O}(n + tl)$ for $\mathrm{NB}_1(\boldsymbol{x})$, $\mathrm{O}(n'tl)$ for $\mathrm{NB}_2(\boldsymbol{x})\backslash\mathrm{NB}_1(\boldsymbol{x})$ and $\mathrm{O}(n'tl\min\{n, tl\})$ for $\mathrm{NB}_3(\boldsymbol{x})\backslash\mathrm{NB}_2(\boldsymbol{x})$. As $n' \leqslant n$ and $l \leqslant n$ always hold, these orders are not worse than those of naive implementation, and are much better if $n' \ll n$ or $l \ll n$ holds.

In our implementation, we compute $\theta_i^{(\kappa)}(\boldsymbol{x})$ and $\Delta pcost(\boldsymbol{x}, \{j\})$ in O(1) time for each $j \in N$, $i \in M$ and $\kappa = 0, 1, 2, \ldots, l$. For this, all the values of $\sum_{j \in N} a_{ij} x_j$ and $\Delta pcost(\boldsymbol{x}, \{j\})$ are stored in memory. Let θ_i (respectively, π_j) denote the value of $\sum_{j \in N} a_{ij} x_j$ (respectively, $\Delta pcost(\boldsymbol{x}, \{j\})$) stored in memory for the current solution $\boldsymbol{x}$.

Given an initial solution $\boldsymbol{x}$, initializing the values of θ_i for all $i \in M$ is possible in $\mathrm{O}(m + \sum_{j \in N} |S_j| x_j) = \mathrm{O}(m + n't)$ time, by first initializing $\theta_i := 0$ for all $i \in M$ and then executing $\theta_i := \theta_i + 1$ for each $i \in S_j$ and $j \in N$ with $x_j = 1$. When $\boldsymbol{x}$ is modified to $\boldsymbol{x} \updownarrow J$, the values of θ_i are updated in $\mathrm{O}(\sum_{j \in J} |S_j|) = \mathrm{O}(t|J|)$ time in a similar manner. Since $J \leqslant r = 3$ holds, this time complexity is $\mathrm{O}(t)$.

Given an initial solution $\boldsymbol{x}$ and penalty weights p_i for all $i \in M$, π_j are initialized by

$$\pi_j := \begin{cases} -c_j + \sum_{i \in S_j} p_i \theta_i^{(1)}(\boldsymbol{x}) & (\text{if } x_j = 1), \\ c_j - \sum_{i \in S_j} p_i \theta_i^{(0)}(\boldsymbol{x}) & (\text{otherwise}) \end{cases}$$

for all $j \in N$. The required computational time for all $j \in N$ is $\mathrm{O}(\sum_{j \in N} |S_j|) = \mathrm{O}(nt)$. The time for this initialization is negligible, since it is necessary only if $\boldsymbol{x}$ is initialized or weights p_i are changed, which does not happen so often. The values of π_j, $j \in N$, are then updated if $\boldsymbol{x}$ is changed to $\boldsymbol{x} \updownarrow J$. Note that the value of π_j is not changed if one of the following two conditions holds for $j \in N\backslash J$:

(1) $x_j = 1$ and $\theta_i^{(1)}(\boldsymbol{x}) = \theta_i^{(1)}(\boldsymbol{x} \updownarrow J)$ for all $i \in S_j$;
(2) $x_j = 0$ and $\theta_i^{(0)}(\boldsymbol{x}) = \theta_i^{(0)}(\boldsymbol{x} \updownarrow J)$ for all $i \in S_j$.

That is, we update only those π_j relevant to the flip. We call the algorithm to update π_j UPDATE($\boldsymbol{x}, J$), which is described in Fig. 2.

Since $|J| \leqslant r = 3$ holds, the time complexity of Steps 1 and 2 is $\mathrm{O}(t)$. In Step 3, $|\{i \in M \mid \theta_i^{(0)}(\boldsymbol{x}) = 1 \text{ and } \theta_i^{(0)}(\boldsymbol{x} \updownarrow J) = 0\}| = \mathrm{O}(t)$ holds, because such i belongs to $\cup_{j \in J} S_j$. Furthermore, as $|\{j \in N\backslash J | x_j = 0$

Algorithm UPDATE($\boldsymbol{x}, J$)

(The values of π_j, $j \in N$, are updated.)

Step 1 Let $\pi_j := -c_j + \sum_{i \in S_j} p_i \theta_i^{(1)}(\boldsymbol{x} \updownarrow J)$, for all $j \in J$ with $x_j = 0$.

Step 2 Let $\pi_j := c_j - \sum_{i \in S_j} p_i \theta_i^{(0)}(\boldsymbol{x} \updownarrow J)$, for all $j \in J$ with $x_j = 1$.

Step 3 Let $\pi_j := \pi_j + p_i$ for all $i \in M$ with $\theta_i^{(0)}(\boldsymbol{x}) = 1$ and $\theta_i^{(0)}(\boldsymbol{x} \updownarrow J) = 0$, and for all $j \in N \setminus J$ with $x_j = 0$ and $i \in S_j$.

Step 4 Let $\pi_j := \pi_j - p_i$ for all $i \in M$ with $\theta_i^{(0)}(\boldsymbol{x}) = 0$ and $\theta_i^{(0)}(\boldsymbol{x} \updownarrow J) = 1$, and for all $j \in N \setminus J$ with $x_j = 0$ and $i \in S_j$.

Step 5 Let $\pi_j := \pi_j - p_i$ for all $i \in M$ with $\theta_i^{(1)}(\boldsymbol{x}) = 1$ and $\theta_i^{(1)}(\boldsymbol{x} \updownarrow J) = 0$, and for all $j \in N \setminus J$ with $x_j = 1$ and $i \in S_j$.

Step 6 Let $\pi_j := \pi_j + p_i$ for all $i \in M$ with $\theta_i^{(1)}(\boldsymbol{x}) = 0$ and $\theta_i^{(1)}(\boldsymbol{x} \updownarrow J) = 1$, and for all $j \in N \setminus J$ with $x_j = 1$ and $i \in S_j$.

Fig. 2. Algorithm to update π_j.

and $i \in S_j\}| = O(l)$ holds, the time for Step 3 is $O(tl)$. Similarly, Steps 4–6 are computed in $O(tl)$ time. Therefore the total time of UPDATE ($\boldsymbol{x}$, J) is $O(tl)$.

3.2.1. Searching 1-flip neighborhood

We first describe how to search $NB_1(\boldsymbol{x})$. We use the information from the Lagrangian relaxation LR-SCP($\boldsymbol{u}$). It is known that subsets S_j in an optimal solution tend to have small $c_j(\boldsymbol{u})$ values if a good Lagrangian vector $\boldsymbol{u}$ is chosen. An algorithm to find such $\boldsymbol{u}$ was already explained in Section 2.2.

The algorithm to search $NB_1(\boldsymbol{x})$, called 1-FLIP($\boldsymbol{x}$), is as follows. We first search for an improved solution obtainable by flipping an x_j from 0 to 1 by searching a j satisfying $x_j = 0$ and $\pi_j < 0$. If such candidates exist, we choose a j with the minimum $c_j(\boldsymbol{u})$. Otherwise, we search for an improved solution obtainable by flipping an x_j from 1 to 0 by searching a j satisfying $x_j = 1$ and $\pi_j < 0$. Then we update the values of relevant π_j if an improved solution was found. The details of algorithm 1-FLIP($\boldsymbol{x}$) is summarized in Fig. 3.

As the size of $NB_1(\boldsymbol{x})$ is $O(n)$, the time complexity of Steps 1 and 2 is $O(n)$. If a better solution is found, π_j is updated by calling UPDATE($\boldsymbol{x}$, $\{j^*\}$) in $O(tl)$ time in Step 3. Therefore the computational time of algorithm 1-FLIP($\boldsymbol{x}$) is $O(n + tl)$.

Remark. In Step 2 of algorithm 1-FLIP($\boldsymbol{x}$), we choose j^* randomly. As we choose a j^* that minimizes $c_j(\boldsymbol{u})$ in Step 1, a natural rule in Step 2 would be to choose a j^* that maximizes $c_j(\boldsymbol{u})$ (i.e., a symmetric rule as in Step 1). We also tried this rule, but it did not work well. The motivation of randomly choosing j^* is explained as follows. Step 2 is executed when $\{j \in N | x_j = 0,\ \pi_j < 0\} = \emptyset$ and $\{j \in N | x_j = 1,\ \pi_j < 0\} \neq \emptyset$ hold. This situation usually happens just after the penalty weights p_i are reduced by the adjustment algorithm in Section 4. In such a case, the penalty weights are reduced by relatively large amounts to diversify the search. Our rule was adopted to help the diversification of the search.

3.2.2. Searching 2-flip neighborhood

To search neighborhood $NB_2(\boldsymbol{x})\backslash NB_1(\boldsymbol{x})$, we derive conditions that reduce the number of candidates without sacrificing the solution quality. Our algorithm is based on the following two lemmas.

Lemma 1. *Suppose that a solution $\boldsymbol{x}$ is locally optimal with respect to $NB_1(\boldsymbol{x})$. Then $\Delta pcost(\boldsymbol{x},\{j_1,j_2\}) < 0$ holds only if $x_{j_1} \neq x_{j_2}$.*

Proof. See Appendix A. □

Lemma 2. *Suppose that $\Delta pcost(\boldsymbol{x},\{j_1\}) \geqslant 0$, $\Delta pcost(\boldsymbol{x},\{j_2\}) \geqslant 0$ and $x_{j_1} \neq x_{j_2}$ hold. Then $\Delta pcost(\boldsymbol{x},\{j_1,j_2\}) < 0$ holds only if $\{i \in S_{j_1} \cap S_{j_2} \mid \theta_i^{(1)}(\boldsymbol{x}) = 1\} \neq \emptyset$.*

Algorithm 1-FLIP($\boldsymbol{x}$)

Input: a solution $\boldsymbol{x}$.

Output: a solution $\boldsymbol{x}' \in NB_1(\boldsymbol{x})$ with $pcost(\boldsymbol{x}') < pcost(\boldsymbol{x})$ if such a solution exists; $\boldsymbol{x}$ otherwise.

Step 1 If $\{j \in N \mid x_j = 0,\ \pi_j < 0\} = \emptyset$, proceed to Step 2. Otherwise choose a $j^* \in \{j \in N \mid x_j = 0,\ \pi_j < 0\}$ with the minimum $c_j(\boldsymbol{u})$, and proceed to Step 3.

Step 2 If $\{j \in N \mid x_j = 1,\ \pi_j < 0\} = \emptyset$, output $\boldsymbol{x}$ and exit. Otherwise randomly choose $j^* \in \{j \in N \mid x_j = 1,\ \pi_j < 0\}$.

Step 3 Update π_j by calling UPDATE($\boldsymbol{x}$, $\{j^*\}$), update θ_i, and let $x_{j^*} := 1 - x_{j^*}$. Output $\boldsymbol{x}$ and exit.

Fig. 3. Algorithm to search $NB_1(\boldsymbol{x})$.

Algorithm 2-FLIP($\boldsymbol{x}$)

Input: a solution $\boldsymbol{x}$.

Output: a solution $\boldsymbol{x}' \in \mathrm{NB}_2(\boldsymbol{x})$ with $pcost(\boldsymbol{x}') < pcost(\boldsymbol{x})$ if such a solution exists; $\boldsymbol{x}$ otherwise.

Step 1 Let $X := \{j \in N \mid x_j = 1\}$.

Step 2 If $X = \emptyset$, output $\boldsymbol{x}$ and exit. Otherwise randomly choose a $\tilde{\jmath} \in X$.

Step 3 Let $C_{\tilde{\jmath}} := \left\{ j \in N \,\middle|\, \{i \in S_{\tilde{\jmath}} \cap S_j \mid \theta_i^{(1)}(\boldsymbol{x}) = 1\} \neq \emptyset \text{ and } x_j = 0 \right\}$ and compute $\Delta pcost(\boldsymbol{x}, \{\tilde{\jmath}, j\})$ for all $j \in C_{\tilde{\jmath}}$.

Step 4 If $\{j \in C_{\tilde{\jmath}} \mid \Delta pcost(\boldsymbol{x}, \{\tilde{\jmath}, j\}) < 0\} = \emptyset$, then let $X := X \setminus \{\tilde{\jmath}\}$ and return to Step 2. Otherwise choose a $\hat{\jmath} \in \{j \in C_{\tilde{\jmath}} \mid \Delta pcost(\boldsymbol{x}, \{\tilde{\jmath}, j\}) < 0\}$ with the minimum $c_j(\boldsymbol{u})$.

Step 5 Update π_j by calling UPDATE($\boldsymbol{x}, \{\tilde{\jmath}, \hat{\jmath}\}$), update θ_i, and let $x_{\tilde{\jmath}} := 0$ and $x_{\hat{\jmath}} := 1$. Output $\boldsymbol{x}$ and exit.

Fig. 4. Algorithm to search $\mathrm{NB}_2(\boldsymbol{x})$.

Proof. See Appendix B. □

By Lemmas 1 and 2, the search in $\mathrm{NB}_2(\boldsymbol{x})\backslash\mathrm{NB}_1(\boldsymbol{x})$ can be restricted to solutions $\boldsymbol{x}\updownarrow\{j_1, j_2\}$ satisfying $x_{j_1} \neq x_{j_2}$ and $\{i \in S_{j_1} \cap S_{j_2} \mid \theta_i^{(1)}(\boldsymbol{x}) = 1\} \neq \emptyset$. Therefore we only generate such candidates and evaluate their *pcost*. The algorithm to search $\mathrm{NB}_2(\boldsymbol{x})\backslash\mathrm{NB}_1(\boldsymbol{x})$, called 2-FLIP($\boldsymbol{x}$), is formally described in Fig. 4. Roughly speaking, algorithm 2-FLIP($\boldsymbol{x}$) is realized as follows. For each $\tilde{\jmath}$ with $x_{\tilde{\jmath}} = 1$, we flip $x_{\tilde{\jmath}}$ to 0 temporarily, call UPDATE($\boldsymbol{x}$,$\{\tilde{\jmath}\}$), and then check $\Delta pcost(\boldsymbol{x}, \{\tilde{\jmath}, j\})$ for all j whose π_j have changed during the call to UPDATE($\boldsymbol{x}, \{\tilde{\jmath}\}$).

In algorithm 2-FLIP($\boldsymbol{x}$), the set $\{\{\tilde{\jmath}, j\} \mid \tilde{\jmath} \in X \text{ and } j \in C_{\tilde{\jmath}}\}$ for the X defined in Step 1 includes all pairs of j_1 and j_2 satisfying $x_{j_1} \neq x_{j_2}$ and $\{i \in S_{j_1} \cap S_{j_2} \mid \theta_i^{(1)}(\boldsymbol{x}) = 1\} \neq \emptyset$. Therefore algorithm 2-FLIP($\boldsymbol{x}$) always finds a solution in $\mathrm{NB}_2(\boldsymbol{x})\backslash\mathrm{NB}_1(\boldsymbol{x})$ that improves $pcost(\boldsymbol{x})$, if such a solution exists.

We now evaluate the time complexity of algorithm 2-FLIP($\boldsymbol{x}$). Since the number of subsets covering an element $i \in S_{\tilde{\jmath}}$ is $\mathrm{O}(l)$ and $|S_{\tilde{\jmath}}| \leqslant t$ holds, we obtain $|C_{\tilde{\jmath}}| = \mathrm{O}(\min\{n, tl\})$. For all $\tilde{\jmath}$ and j, we have

$$\Delta pcost(\boldsymbol{x}\updownarrow\{\tilde{\jmath}, j\}) = \Delta pcost(\boldsymbol{x}, \{\tilde{\jmath}\}) + \Delta pcost(\boldsymbol{x}\updownarrow\{\tilde{\jmath}\}, \{j\}).$$

Since $\Delta pcost(\boldsymbol{x}, \{\tilde{\jmath}\})$ is stored in $\pi_{\tilde{\jmath}}$, it is retrieved in O(1) time. Then $\Delta pcost(\boldsymbol{x}\updownarrow\{\tilde{\jmath}\}, \{j\})$ for all $j \in C_{\tilde{\jmath}}$ is computed in $\mathrm{O}(tl)$ time by using algorithm UPDATE($\boldsymbol{x}, \{\tilde{\jmath}\}$) (i.e., $\Delta pcost(\boldsymbol{x}\updownarrow\{\tilde{\jmath}\}, \{j\}) = \pi_j$ holds for all $j \in N$ after applying UPDATE($\boldsymbol{x}, \{\tilde{\jmath}\}$)). Therefore, the time to compute $\Delta pcost(\boldsymbol{x}, \{\tilde{\jmath}, j\})$ for all $j \in C_{\tilde{\jmath}}$ in Step 3 is $\mathrm{O}(tl)$. The computational time in Step 4 is $\mathrm{O}(tl)$ since $|C_{\tilde{\jmath}}| = \mathrm{O}(tl)$ holds. After Step 4, we update the values of π_j, $j \in N$, to $\Delta pcost(\boldsymbol{x}, \{j\})$ by using UPDATE($\boldsymbol{x}\updownarrow\{\tilde{\jmath}\}, \{\tilde{\jmath}\}$), which takes $\mathrm{O}(tl)$ time. The time to update π_j with UPDATE($\boldsymbol{x}, \{\tilde{\jmath}, \hat{\jmath}\}$) in Step 5 is $\mathrm{O}(tl)$. As $|X| = \mathrm{O}(n')$ holds for the X in Step 1, the number of repetitions of Steps 2–4 is $\mathrm{O}(n')$. Therefore the total time of algorithm 2-FLIP($\boldsymbol{x}$) is $\mathrm{O}(n'tl)$.

3.2.3. Searching 3-flip neighborhood

To search $\mathrm{NB}_3(\boldsymbol{x})\backslash\mathrm{NB}_2(\boldsymbol{x})$ efficiently, we first derive conditions that reduce the number of candidates without sacrificing the solution quality.

Lemma 3. *Suppose that $\boldsymbol{x}$ is locally optimal with respect to $\mathrm{NB}_1(\boldsymbol{x})$. Then $\Delta pcost(\boldsymbol{x}, \{j_1, j_2, j_3\}) < 0$ holds only if $x_{j_1} \neq x_{j_2}$ or $x_{j_2} \neq x_{j_3}$ holds.*

Proof. See Appendix C. □

Lemma 4. *Suppose that* $\boldsymbol{x}$ *is locally optimal with respect to* $\mathrm{NB}_2(\boldsymbol{x})$, *and* $x_{j_1} \neq x_{j_2}$ *or* $x_{j_2} \neq x_{j_3}$ *holds. We assume* $x_{j_1} = 1$ *and* $x_{j_2} = 0$ *without loss of generality. Then* $\Delta pcost(\boldsymbol{x}, \{j_1, j_2, j_3\}) < 0$ *holds only if one of the following two conditions holds*:

(1) $x_{j_3} = 1$, $\{i \in S_{j_1} \cap S_{j_2} \mid \theta_i^{(1)}(\boldsymbol{x}) = 1\} \neq \emptyset$ *and* $\{i \in S_{j_2} \cap S_{j_3} \mid \theta_i^{(1)}(\boldsymbol{x}) = 1\} \neq \emptyset$;
(2) $x_{j_3} = 0$, $\{i \in S_{j_1} \cap S_{j_2} \setminus S_{j_3} \mid \theta_i^{(1)}(\boldsymbol{x}) = 1\} \neq \emptyset$ *and* $\{i \in S_{j_1} \cap S_{j_3} \setminus S_{j_2} \mid \theta_i^{(1)}(\boldsymbol{x}) = 1\} \neq \emptyset$.

Proof. See Appendix D. □

By Lemmas 3 and 4, the search in $\mathrm{NB}_3(\boldsymbol{x}) \backslash \mathrm{NB}_2(\boldsymbol{x})$ can be restricted to the solutions $\boldsymbol{x} \updownarrow \{j_1, j_2, j_3\}$ satisfying one of the conditions (1) and (2) of Lemma 4. Therefore we only generate such solutions and evaluate their *pcost*. The algorithm to search $\mathrm{NB}_3(\boldsymbol{x}) \backslash \mathrm{NB}_2(\boldsymbol{x})$, called 3-FLIP($\boldsymbol{x}$), is formally described in Fig. 5. The algorithm may seem complicated, but the basic idea of realizing it is similar to algorithm 2-FLIP($\boldsymbol{x}$) and is roughly explained as follows. For each $\tilde{\jmath}$ with $x_{\tilde{\jmath}} = 1$, we flip $x_{\tilde{\jmath}}$ to 0 temporarily and call UPDATE($\boldsymbol{x}, \{\tilde{\jmath}\}$); then flip $x_{j'}$ with $x'_j = 0$ to 1 temporarily and call UPDATE($\boldsymbol{x} \updownarrow \{\tilde{\jmath}\}, \{j'\}$) for each j' whose π'_j has changed during the call to UPDATE($\boldsymbol{x}, \{\tilde{\jmath}\}$); and then check $\Delta pcost(\boldsymbol{x}, \{\tilde{\jmath}, j', j\})$ for all j whose π_j have changed during the call to UPDATE($\boldsymbol{x} \updownarrow \{\tilde{\jmath}\}, \{j'\}$).

In algorithm 3-FLIP($\boldsymbol{x}$), the set $\{\{\tilde{\jmath}, j', j\} \mid \tilde{\jmath} \in X, j' \in C_{\tilde{\jmath}} \text{ and } j \in D_{j'}\}$ (for the X defined in Step 1 and the $C_{\tilde{\jmath}}$ defined in Step 3) includes the set of $\{j_1, j_2, j_3\}$ that satisfy condition (1) of Lemma 4. Similarly the set $\{\{\tilde{\jmath}, j', j\} \mid \tilde{\jmath} \in X, j' \in C_{\tilde{\jmath}} \text{ and } j \in C_{\tilde{\jmath}}\}$ includes the set of $\{j_1, j_2, j_3\}$ that satisfy condition (2) of Lemma 4. Therefore algorithm 3-FLIP($\boldsymbol{x}$) always finds a solution in $\mathrm{NB}_3(\boldsymbol{x}) \backslash \mathrm{NB}_2(\boldsymbol{x})$ that improves *pcost*($\boldsymbol{x}$), if such a solution exists.

We now consider the computational time of algorithm 3-FLIP($\boldsymbol{x}$). As in Section 3.2.2, $|X| = \mathrm{O}(n')$ and $|C_{\tilde{\jmath}}| = \mathrm{O}(\min\{n, tl\})$ hold. For any $\tilde{\jmath}$, j' and j, we have

$$\Delta pcost(\boldsymbol{x}\ , \{\tilde{\jmath}, j', j\}) = \Delta pcost(\boldsymbol{x}, \{\tilde{\jmath}\}) + \Delta pcost(\boldsymbol{x} \updownarrow \{\tilde{\jmath}\}, \{j'\}) + \Delta pcost(\boldsymbol{x} \updownarrow \{\tilde{\jmath}, j'\}, \{j\}).$$

Algorithm 3-FLIP($\boldsymbol{x}$)

Input: a solution $\boldsymbol{x}$.

Output: a solution $\boldsymbol{x'} \in \mathrm{NB}_3(\boldsymbol{x})$ with $pcost(\boldsymbol{x'}) < pcost(\boldsymbol{x})$ if such a solution exists; $\boldsymbol{x}$ otherwise.

Step 1 Let $X := \{j \in N \mid x_j = 1\}$.

Step 2 If $X = \emptyset$, output $\boldsymbol{x}$ and exit. Otherwise randomly choose a $\tilde{\jmath} \in X$.

Step 3 Let $C_{\tilde{\jmath}} := \left\{ j \in N \,\middle|\, \{i \in S_{\tilde{\jmath}} \cap S_j \mid \theta_i^{(1)}(\boldsymbol{x}) = 1\} \neq \emptyset \text{ and } x_j = 0 \right\}$.

Step 4 If $C_{\tilde{\jmath}} = \emptyset$, let $X := X \setminus \{\tilde{\jmath}\}$ and return to Step 2. Otherwise randomly choose a $j' \in C_{\tilde{\jmath}}$.

Step 5 Let $D_{j'} := \left\{ j \in N \,\middle|\, \{i \in S_{j'} \cap S_j \mid \theta_i^{(1)}(\boldsymbol{x}) = 1\} \neq \emptyset,\ x_j = 1 \right\}$ and compute $\Delta pcost(\boldsymbol{x}, \{\tilde{\jmath}, j', j\})$ for all $j \in C_{\tilde{\jmath}} \cup D_{j'}$.

Step 6 Choose a $\hat{\jmath} \in C_{\tilde{\jmath}} \cup D_{j'}$ with the minimum $\Delta pcost(\boldsymbol{x}, \{\tilde{\jmath}, j', j\})$. If $\Delta pcost(\boldsymbol{x}, \{\tilde{\jmath}, j', \hat{\jmath}\}) < 0$, proceed to Step 7, otherwise let $C_{\tilde{\jmath}} := C_{\tilde{\jmath}} \setminus \{j'\}$ and return to Step 4.

Step 7 Update π_j by calling UPDATE($\boldsymbol{x}, \{\tilde{\jmath}, j', \hat{\jmath}\}$), and let $x_j := 1 - x_j$ for all $j \in \{\tilde{\jmath}, j', \hat{\jmath}\}$. Output $\boldsymbol{x}$ and exit.

Fig. 5. Algorithm to search $\mathrm{NB}_3(\boldsymbol{x})$.

Since $\Delta pcost(\boldsymbol{x}, \{\tilde{\jmath}\})$ is stored in $\pi_{\tilde{\jmath}}$, it is retrieved in O(1) time. Then $\Delta pcost(\boldsymbol{x} \updownarrow \{\tilde{\jmath}\}, \{j'\})$ and $\Delta pcost(\boldsymbol{x} \updownarrow \{\tilde{\jmath}, j'\}, \{j\})$ for all $j \in C_{\tilde{\jmath}} \cup D_{j'}$ are computed in O(tl) time by calling UPDATE($\boldsymbol{x}, \{\tilde{\jmath}\}$) and UPDATE($\boldsymbol{x} \updownarrow \{\tilde{\jmath}\}, \{j'\}$) (i.e., $\Delta pcost(\boldsymbol{x} \updownarrow \{\tilde{\jmath}\}, \{j'\}) = \pi_{j'}$ holds for all $j \in N$ after applying UPDATE($\boldsymbol{x}, \{\tilde{\jmath}\}$), and $\Delta pcost(\boldsymbol{x} \updownarrow \{\tilde{\jmath}, j'\}, \{j\}) = \pi_j$ holds for all $j \in N$ after applying both UPDATE($\boldsymbol{x}, \{\tilde{\jmath}\}$) and UPDATE($\boldsymbol{x} \updownarrow \{\tilde{\jmath}\}, \{j'\}$)). Therefore the time to compute $\Delta pcost(\boldsymbol{x}, \{\tilde{\jmath}, j', j\})$ in Step 5 is O(tl). Computational time in Step 6 is O(tl), since $|C_{\tilde{\jmath}} \cup D'_j| = O(tl)$ holds. After Step 6, we update all π_j to $\Delta pcost(\boldsymbol{x}, \{j\})$ by calling UPDATE($\boldsymbol{x} \updownarrow \{\tilde{\jmath}, j'\}, \{\tilde{\jmath}, j'\}$), which takes O($tl$) time. The time to update π_j by calling UPDATE($\boldsymbol{x}, \{\tilde{\jmath}, j', \hat{\jmath}\}$) in Step 7 is O($tl$). Therefore the total time of algorithm 3-FLIP($\boldsymbol{x}$) is $O(n'tl \min\{n, tl\})$.

3.3. Additional rules to speed up the local search

Though the local search with a large neighborhood is effective in finding better solutions, the computational time often becomes excessive even after the reduction of Section 3.2. Therefore the neighborhood size is further reduced depending on the current solution $\boldsymbol{x}$. Let LB be the lower bound for the objective value of SCP obtained from Lagrangian relaxation LR-SCP($\boldsymbol{u}$). We restrict the neighborhood to $NB_2(\boldsymbol{x})$ if $cost(\boldsymbol{x}) <$ LB holds (in this case $\boldsymbol{x}$ is always infeasible); otherwise we use $NB_3(\boldsymbol{x})$. In case of $cost(\boldsymbol{x}) <$ LB, $\boldsymbol{x}$ is usually far from the feasible region and it is unlikely that good feasible solutions exist around $\boldsymbol{x}$.

Furthermore we forbid the search to visit solutions $\boldsymbol{x} \in \{0,1\}^n$ with $cost(\boldsymbol{x}) \geqslant$ UB, where UB is the incumbent value (i.e., the cost of the best feasible solution obtained during the search so far), since the possibility of finding a better feasible solution in such region seems small.

As a result of these additional rules, the obtained solution may not be locally optimal. However, it was observed in our experiment that the solution quality did not degrade much and the overall performance (in terms of both solution quality and computational time) was improved, since more iterations of local search become possible by such speed up.

4. Adaptive control of penalty weights

Let $\boldsymbol{x}^{\text{prev}}$ denote the solution at which the previous local search stops. In our algorithm, when the local search stops at solution $\boldsymbol{x}^{\text{prev}}$, it resumes from $\boldsymbol{x}^{\text{prev}}$ after updating penalty weights p_i (in order to realize strategic oscillation). Starting from the following initial values:

$$p_i := \min\{c_j \mid i \in S_j\} \quad \forall i \in M, \tag{1}$$

the weights p_i are updated as follows. If

$$cost(\boldsymbol{x}^{\text{prev}}) < \text{UB} \quad \text{and} \quad \{\boldsymbol{x} \in NB_1(\boldsymbol{x}^{\text{prev}}) \mid pcost(\boldsymbol{x}) < pcost(\boldsymbol{x}^{\text{prev}}) \text{ and } cost(\boldsymbol{x}) \geqslant \text{UB}\} = \emptyset \tag{2}$$

hold, p_i are updated by

$$p_i := p_i(1 + \theta_i^{(0)}(\boldsymbol{x}^{\text{prev}}) \max\{\delta^+(\boldsymbol{x}^{\text{prev}}), \varepsilon^+\}) \quad \forall i \in M. \tag{3}$$

Otherwise, p_i are updated by

$$p_i := p_i(1 - \max\{\delta^-(\boldsymbol{x}^{\text{prev}}), \varepsilon^-\}) \quad \forall i \in M. \tag{4}$$

Here ε^+ (> 0) and ε^- (> 0) are program parameters, and $\delta^+(\boldsymbol{x}^{\text{prev}})$ and $\delta^-(\boldsymbol{x}^{\text{prev}})$ are the step sizes defined by the following rules. Let

$$\delta^+_{NB_1}(\boldsymbol{x}^{\text{prev}}) = \min\left\{ \frac{\Delta pcost(\boldsymbol{x}^{\text{prev}}, \{j\})}{\sum_{i \in S_j} p_i \theta_i^{(0)}(\boldsymbol{x}^{\text{prev}})} \,\middle|\, x_j^{\text{prev}} = 0 \quad \text{and} \quad \sum_{i \in S_j} p_i \theta_i^{(0)}(\boldsymbol{x}^{\text{prev}}) > 0 \right\}$$

$$\delta^+_{\mathrm{NB}_2}(\boldsymbol{x}^{\mathrm{prev}}) = \min\left\{ \frac{\Delta pcost(\boldsymbol{x}^{\mathrm{prev}}, \{j_1, j_2\})}{\sum_{i \in S_{j_2}} p_i \theta_i^{(0)}(\boldsymbol{x}^{\mathrm{prev}})} \middle| x_{j_1}^{\mathrm{prev}} = 1, \quad x_{j_2}^{\mathrm{prev}} = 0, \quad \sum_{i \in S_{j_2}} p_i \theta_i^{(0)}(\boldsymbol{x}^{\mathrm{prev}}) > 0 \right.$$
$$\left. \text{and} \quad \Delta pcost(\boldsymbol{x}^{\mathrm{prev}}, \{j_1, j_2\}) \geqslant 0 \right\}$$

and $\delta^+(\boldsymbol{x}^{\mathrm{prev}})$ is given by

$$\delta^+(\boldsymbol{x}^{\mathrm{prev}}) := (1 + \beta^+) \min\{\delta^+_{\mathrm{NB}_1}(\boldsymbol{x}^{\mathrm{prev}}), \delta^+_{\mathrm{NB}_2}(\boldsymbol{x}^{\mathrm{prev}})\},$$

where β^+ (> 0) is a program parameter. Let $j_k \in N$ ($k = 1, 2, \ldots, n'$) be those indices satisfying $x_{j_k}^{\mathrm{prev}} = 1$. Then let

$$\delta^-_k(\boldsymbol{x}^{\mathrm{prev}}) = \left\{ \frac{\Delta pcost(\boldsymbol{x}^{\mathrm{prev}}, \{j_k\})}{\sum_{i \in S_{j_k}} p_i \theta_i^{(1)}(\boldsymbol{x}^{\mathrm{prev}})} \right\}, \quad k = 1, 2, \ldots, n',$$

where j_k are ordered so that

$$\delta^-_1(\boldsymbol{x}^{\mathrm{prev}}) \leqslant \delta^-_2(\boldsymbol{x}^{\mathrm{prev}}) \leqslant \cdots \leqslant \delta^-_{n'}(\boldsymbol{x}^{\mathrm{prev}})$$

holds. Then $\delta^-(\boldsymbol{x}^{\mathrm{prev}})$ is defined by

$$\delta^-(\boldsymbol{x}^{\mathrm{prev}}) = \min\{(1 + \beta^-)\delta^-_{k'}(\boldsymbol{x}^{\mathrm{prev}}), 1 - \varepsilon^-\}, \quad k' = \min\{\eta, n'\}, \tag{5}$$

where β^- (> 0) and η (a positive integer) are program parameters. Since $\delta^-(\boldsymbol{x}^{\mathrm{prev}})$ is chosen so that $\delta^-(\boldsymbol{x}^{\mathrm{prev}}) < 1$ holds and the initial values of p_i ($\forall i \in M$) are positive by definition (1), $p_i > 0$ holds for all $i \in M$ during the search.

This value of $\delta^+(\boldsymbol{x}^{\mathrm{prev}})$ is chosen so that $|\{\boldsymbol{x}' \in \mathrm{NB}_2(\boldsymbol{x}^{\mathrm{prev}}) \mid pcost(\boldsymbol{x}') < pcost(\boldsymbol{x}^{\mathrm{prev}})\}| \geqslant 1$ holds for the new values of p_i, and the value of $\delta^-(\boldsymbol{x}^{\mathrm{prev}})$ is basically chosen so that

$$|\{j \in N \mid \Delta pcost(\boldsymbol{x}^{\mathrm{prev}}, \{j\}) < 0, x_j^{\mathrm{prev}} = 1\}| \geqslant k'$$

holds for the new values of p_i (see Appendix E). By these rules, the search from $\boldsymbol{x}^{\mathrm{prev}}$ always moves to other solutions after updating penalty weights. We set the above program parameter values to

$$\varepsilon^+ = 0.05, \ \varepsilon^- = 0.01, \ \beta^+ = 0.1, \ \beta^- = 0.1 \quad \text{and} \quad \eta = 10$$

in our experiment, to ensure enough changes of penalty weights.

We judge by condition (2) whether the current penalty weights are large enough to obtain feasible solutions or not. If (2) holds, we increase p_i for all uncovered $i \in M$, because there is a possibility of finding feasible solutions which improve UB. Otherwise, even if penalty weights are increased, most solutions $\boldsymbol{x}$ in $\mathrm{NB}_3(\boldsymbol{x}^{\mathrm{prev}})$ with better $pcost(\boldsymbol{x}^{\mathrm{prev}})$ will not satisfy $cost(\boldsymbol{x}) < \mathrm{UB}$. In this sense, solutions around $\boldsymbol{x}^{\mathrm{prev}}$ are not promising. Therefore, we decrease p_i for all $i \in M$ rather sharply to force the search away from $\boldsymbol{x}^{\mathrm{prev}}$. The algorithm to update the penalty weights p_i is summarized in Fig. 6.

Algorithm UPDATE-PENALTY($\boldsymbol{x}^{\mathrm{prev}}$)

Step 1 If condition (2) holds, proceed to Step 2; otherwise, proceed to Step 3.

Step 2 Increase p_i by (3). Output p_i for all $i \in M$ and exit.

Step 3 Decrease p_i by (4). Output p_i for all $i \in M$ and exit.

Fig. 6. Algorithm to update penalty weights.

5. Heuristic reduction of problem sizes

In this section, we describe how to reduce the size of problem instances by heuristically fixing some variables x_j to 0 or 1. This size reduction is applied many times, before starting the local search and during the iterations of the local search. In other words, the set of fixed variables are modified dynamically by using the information from the Lagrangian relaxation.

We define a set of indices $N_1 \subset N$ (respectively, $N_0 \subset N$), such that variables $x_j, j \in N_1$ (respectively, N_0), are fixed to 1 (respectively, 0), where $N_1 \cap N_0 = \emptyset$, and we let $N_{\text{free}} = N \setminus (N_0 \cup N_1)$ be the set of indices of free variables. Given $(N_1, N_0, N_{\text{free}})$, we define the following problem:

$$\begin{aligned} \text{SCP}(N_1, N_0, N_{\text{free}}): \quad & \text{minimize} \quad cost(\boldsymbol{x}) = \sum_{j \in N_{\text{free}}} c_j x_j \\ & \text{subject to} \quad \sum_{j \in N_{\text{free}}} a_{ij} x_j \geqslant 1 \quad \forall i \in M \quad \text{such that} \quad \sum_{j \in N_1} a_{ij} = 0 \\ & \qquad x_j \in \{0,1\} \quad \forall j \in N_{\text{free}}. \end{aligned}$$

The problem $\text{SCP}(N_1, N_0, N_{\text{free}})$ is also an SCP, with variables $x_j, j \in N_{\text{free}}$, whose size is smaller than the original instance. Redundant constraints $i \in M$ with $\sum_{j \in N_1} a_{ij} \geqslant 1$ are also removed in the reduced problem.

In Sections 5.1 and 5.2, we explain how to choose $(N_1, N_0, N_{\text{free}})$. In Section 5.1, we consider the first stage of fixing variables, which is applied before starting the local search. In Section 5.2, we consider the adaptive modification of the fixed variables.

5.1. First stage of fixing variables

Before starting the local search, we choose a partition $(N_1, N_0, N_{\text{free}})$. The choice of variables to be fixed is based on the relative costs $c_j(\boldsymbol{u}^+)$, where $\boldsymbol{u}^+$ is the Lagrangian multiplier vector obtained by the first call of SUBGRADIENT(UB, $\boldsymbol{u}^{(0)}$). It is known that each subset S_j in an optimal solution of SCP tends to have a small $c_j(\boldsymbol{u})$ value if a good Lagrangian multiplier vector $\boldsymbol{u}$ is used (e.g., [10]). Therefore, in our algorithm, we choose N_{free} to be the set of indices with small $c_j(\boldsymbol{u}^+)$ values, and we then set $N_1 := \emptyset$ and $N_0 := N \setminus N_{\text{free}}$. The procedure is described in Fig. 7, where α ($\geqslant 1$) and *min_free* (a positive integer) are the program parameters used to determine the size of N_{free}. We set

$$\alpha = 3 \quad \text{and} \quad min\_free = 100$$

in the computational experiments of Section 7.

5.2. Modification stage

After finishing some iterations of local search, we modify $(N_1, N_0, N_{\text{free}})$ by first randomly choosing a new set N_1 and then using the information from the resulting Lagrangian relaxation of problem

Algorithm FIRST-FIXING(u^+)

Step 1 Let $\boldsymbol{x} :=$ GREEDY and $K := \min\left[\max\left\{\alpha \sum_{j \in N} x_j, min\_free\right\}, n\right]$.

Step 2 Let N_{free} ($\subseteq N$) be the set of indices of K smallest $c_j(\boldsymbol{u}^+)$.

Step 3 For each $i \in M$, if $i \notin \cup_{j \in N_{\text{free}}} S_j$, choose the j with the minimum $c_j(\boldsymbol{u}^+)$ from those satisfying $i \in S_j$, and add it to N_{free}.

Step 4 Let $N_1 := \emptyset$ and $N_0 := N \setminus N_{\text{free}}$. Output $(N_1, N_0, N_{\text{free}})$ and exit.

Fig. 7. Algorithm for the first fixing.

SCP($N_1, N_0, N_{\text{free}}$). The new set N_1 is randomly chosen from those indices which are included in both the incumbent solution (i.e., the best feasible solution obtained during the search so far) and the current solution. Since better solutions may exist around the incumbent solution, we choose N_1 from the indices included in the incumbent solution. The reason to restrict N_1 to be included also in the current solution is to resume the local search from the current solution. In this modification stage, some elements in N_0 are changed back to be free (i.e., N_0 decreases monotonically during the modification algorithm).

Let $\boldsymbol{x}^*$ be the incumbent solution, $\boldsymbol{x}$ be the current solution, and $\boldsymbol{u}^+$ be the Lagrangian multiplier vector obtained by the first call of SUBGRADIENT(UB, $\boldsymbol{u}^{(0)}$). The procedure of modifying ($N_1, N_0, N_{\text{free}}$) in this stage is described in Fig. 8. In the algorithm, ξ (> 0) is a parameter used to determine the size of N_1, where we set

$$\xi = 0.5$$

in the computational experiments of Section 7.

The frequency of calling MODIFY-FIXING($\boldsymbol{x}^*, \boldsymbol{x}, \boldsymbol{u}^+, N_1, N_0, N_{\text{free}}$) has a large influence on the performance of our algorithm. Calls to MODIFY-FIXING consume much computational time, but infrequent applications may result in insufficient diversification of the search. We execute MODIFY-FIXING whenever both of the following two conditions are satisfied.

(1) The penalty weights are decreased by rule (4) in Section 4.
(2) The local search is iterated at least *minitr_ls* (a prespecified integer) times after the last call of MODIFY-FIXING or after the incumbent solution $\boldsymbol{x}^*$ is improved.

It is observed in a preliminary experiment that the performance is sensitive to the parameter value of *minitr_ls* and its appropriate values are

10–50 for instances with $n = 5000$–$10{,}000$,

100–500 for instances with $n = 50{,}000$–$1{,}000{,}000$.

Algorithm MODIFY-FIXING($\boldsymbol{x}^*$, $\boldsymbol{x}$, $\boldsymbol{u}^+$, $N_1, N_0, N_{\text{free}}$)

Step 1 Let $V := \{ j \in N \mid x^*_j = x_j = 1 \}$, $N_1 := \emptyset$ and $c_{\max} := \max_{j \in V}\{c_j(\boldsymbol{u}^+)\}$.

Step 2 If $V = \emptyset$ or $|\cup_{j \in N_1} S_j|/m \geq \xi$, go to Step 4.

Step 3 Randomly choose a j from set V, where the probability of choosing $j \in V$ is

$$f_j(\boldsymbol{u}^+) = \frac{c_{\max} - c_j(\boldsymbol{u}^+)}{\sum_{\hat{j} \in V} \left(c_{\max} - c_{\hat{j}}(\boldsymbol{u}^+)\right)}.$$

(Uniform distribution is used if $\sum_{\hat{j} \in V}(c_{\max} - c_{\hat{j}}(\boldsymbol{u}^+)) = 0$ holds.) Then let $N_1 := N_1 \cup \{j\}$ and $V := V \setminus \{j\}$, and return to Step 2.

Step 4 Let $\tilde{\boldsymbol{u}} :=$ SUBGRADIENT(UB, $\boldsymbol{u}^+$) (applied to the Lagrangian relaxation of SCP($N_1, \emptyset, N \setminus N_1$)). Let

$$u'_i := \begin{cases} \tilde{u}_i & (\text{if } i \notin \cup_{j \in N_1} S_j) \\ 0 & (\text{otherwise}), \end{cases}$$

and compute $c_j(\boldsymbol{u}')$ for all $j \in N \setminus N_1$.

Step 5 Let $N_0 := N_0 \setminus \{j \in N \mid c_j(\boldsymbol{u}') \leq 0\}$ and $N_{\text{free}} := N \setminus (N_1 \cup N_0)$. Output ($N_1, N_0, N_{\text{free}}$) and exit.

Fig. 8. Algorithm to modify fixing.

Based on these, we used

$$minitr\_ls = 100$$

in our experiment of Section 7.

5.3. Logical tests

Among well-known logical tests for reducing the size of an instance [5,15], we use the following simple dominance rule:

$$\text{If} \quad S_{j_1} \subseteq S_{j_2} \quad \text{and} \quad c_{j_1} \geqslant c_{j_2}, \quad \text{then we can fix} \quad x_{j_1} := 0. \tag{6}$$

Though the rule is simple, it is very expensive if we check the above dominance for all pairs of j_1 and j_2 in N. We therefore use this in a limited way. When a j_1 is to be added into N_{free} in algorithm FIRST-FIXING or MODIFY-FIXING, we check the condition of rule (6) for the j_1 and all j_2 already in $N_1 \cup N_{\text{free}}$. If appropriately implemented, this check is possible in $O(|S_{j_1}|l')$ time for a j_1 and all $j_2 \in N_1 \cup N_{\text{free}}$, where $l' = \max\{\sum_{j \in N_1 \cup N_{\text{free}}} a_{ij} \mid i \in M\}$.

6. Framework of the entire algorithm

The algorithm proposed in this paper consists of two phases, the phase of local search and the phase of fixing variables, which are repeated alternately until a stopping criterion is satisfied. In the local search phase, solutions in the neighborhood are searched according to the rules explained in Section 3. In the variable fixing phase, some variables are fixed as explained in Section 5. The entire algorithm is described in Fig. 9. In the algorithm, *time_lim* is a prespecified limit on the CPU time.

In this algorithm, Step 1 is the initialization, Steps 2 and 9 are the phase of fixing variables (Step 2 is the first stage and Step 9 is the modification stage), and Steps 3–8 are the phase of local search with the adaptive control of penalty weights. Parameter *counter* counts the number of local search iterations applied after the incumbent solution $\boldsymbol{x}^*$ is updated or the fixed variables are modified. Note that one iteration of local search is defined to be the process of finding a solution whose *pcost* cannot be improved by Steps 4–6 (unless the penalty weights are updated in Step 8).

Remark. Though it is not explicitly explained in the steps of Fig. 9, procedures 1-FLIP($\boldsymbol{x}$), 2-FLIP($\boldsymbol{x}$) and 3-FLIP($\boldsymbol{x}$) are applied to problem SCP($N_1, N_0, N_{\text{free}}$), and the Lagrangian multiplier vector $\boldsymbol{u}^*$ obtained in the last call of SUBGRADIENT (UB, $\boldsymbol{u}^{(0)}$) is used in procedures 1-FLIP($\boldsymbol{x}$) and 2-FLIP($\boldsymbol{x}$).

7. Computational experiment

Our algorithm 3-FNLS was evaluated using the benchmark instances obtained electronically from OR-Library. [2] The algorithm was coded in C and run on a workstation Sun Ultra 2 Model 2300 (two Ultra SPARC II 300 MHz processors with 1 GB memory), where the computation was executed on a single processor. Test instances are explained in Section 7.1. In Section 7.2, the effect of neighborhood sizes is compared. In Section 7.3, 3-FNLS is applied to small instances whose exact optimal values are known. In Section 7.4, 3-FNLS is applied to instances called STS, which formulate difficult combinatorial problems.

[2] http://mscmga.ms.ic.ac.uk/jeb/orlib/scpinfo.html

Algorithm 3-FNLS

Input: an instance of SCP.

Output: an approximate solution $\boldsymbol{x}^*$.

Step 1 Let $\boldsymbol{x}^*$:=GREEDY, UB := $cost(\boldsymbol{x}^*)$, $u_i^{(0)} = \min\{ c_j/|S_j| \mid i \in S_j \}$ for all $i \in M$ and $\boldsymbol{u}^+$:= SUBGRADIENT(UB, $\boldsymbol{u}^{(0)}$).

Step 2 Let $(N_1, N_0, N_{\text{free}})$:= FIRST-FIXING($\boldsymbol{u}^+$), *counter* := 0 and $\boldsymbol{x} := \boldsymbol{0}$. Initialize p_i for all $i \in M$ by formula (1) in Section 4.

Step 3 If the computational time exceeds *time_lim*, output $\boldsymbol{x}^*$ and stop.

Step 4 Let $\tilde{\boldsymbol{x}}$:= 1-FLIP($\boldsymbol{x}$). If $\tilde{\boldsymbol{x}} \neq \boldsymbol{x}$, let $\boldsymbol{x} := \tilde{\boldsymbol{x}}$ and return to Step 4.

Step 5 Let $\tilde{\boldsymbol{x}}$:= 2-FLIP($\boldsymbol{x}$). If $\tilde{\boldsymbol{x}} \neq \boldsymbol{x}$, let $\boldsymbol{x} := \tilde{\boldsymbol{x}}$ and return to Step 4.

Step 6 If $cost(\boldsymbol{x}) \leq L(\boldsymbol{u}^+)$ holds, proceed to Step 7. Otherwise let $\tilde{\boldsymbol{x}}$:= 3-FLIP($\boldsymbol{x}$). If $\tilde{\boldsymbol{x}} \neq \boldsymbol{x}$, let $\boldsymbol{x} := \tilde{\boldsymbol{x}}$ and return to Step 4.

Step 7 Let *counter* := *counter* + 1. If feasible solutions $\boldsymbol{x}$ were found in Steps 4, 5 and 6, let $\boldsymbol{x}^+$ be the one with the minimum $cost(\boldsymbol{x})$. If $cost(\boldsymbol{x}^+) <$ UB, let $\boldsymbol{x}^* := \boldsymbol{x}^+$, UB := $cost(\boldsymbol{x}^*)$ and *counter* := 0.

Step 8 Update the penalty weights p_i by calling UPDATE-PENALTY($\boldsymbol{x}$). If the penalty weights are increased or *counter* < *minitr_ls* holds, return to Step 3.

Step 9 Modify $(N_1, N_0, N_{\text{free}})$ by calling MODIFY-FIXING($\boldsymbol{x}^*$, $\boldsymbol{x}$, $\boldsymbol{u}^+$, $N_1, N_0, N_{\text{free}}$). Let *counter* := 0 and return to Step 3.

Fig. 9. The entire algorithm.

We then compare 3-FNLS with other existing heuristic algorithms in Section 7.5 on problem instances of Section 7.1.

7.1. Test instances

There are 13 types of benchmark instances called types 4, 5, 6, A, B, C, D, E, F, G, H, STS and RAIL. The data of these instances are given in Table 1. Each of types 4 and 5 has 10 instances, each of types 6 and A–H has five instances, type STS has four instances and type RAIL has seven instances. Types 4–6 and A–H are randomly generated, where m ranges from 200 to 1000, n ranges from 1000 to 10,000, c_j are random integers from $[1, 100]$, and the density $\sum_{i \in M} \sum_{j \in N} a_{ij}/mn$ ranges from 0.02 to 0.2. For these random instances, optimal solutions are known except for some instances of types E, G and H.

Type STS arose from Steiner triple systems and has regular features, such that $c_j = 1$ for all $j \in N$, $\sum_{j \in N} a_{ij} = 3$ for all $i \in M$ and $|S_{j_1} \cap S_{j_2}| = 1$ for all j_1 and j_2 ($j_1 \neq j_2$). STS instances can be generated recursively from instances called A_3 and A_{15}, where the rule to generate A_{3y} from an instance A_y is found in [18]. A_{135} and A_{243} were taken from OR-Library, and A_{405} and A_{729} were generated by ourselves from A_{135} and A_{243}, respectively. The perl script to generate large STS instances from existing STS instances is obtained from our WWW site. [3] Because of its symmetry, all relative costs $c_j(\boldsymbol{u}^*)$ have the same value for an optimal Lagrangian multiplier vector $\boldsymbol{u}^*$. Therefore the information from the Lagrangian relaxation is useless, and type STS is known to be very difficult.

Type RAIL arose from the railway crew scheduling problem, and includes very large-scale instances, where m ranges from 500 to 5000, n ranges from 50,000 to 1,000,000, c_j are 1 or 2 and the nonzero density

[3] http://www-or.amp.i.kyoto-u.ac.jp/~yagiura/scp/stcp/

Table 1
Details of the test instances

Instance	m	n	Density (%)	Cost range
Type 4	200	1000	2	[1, 100]
Type 5	200	2000	2	[1, 100]
Type 6	200	1000	5	[1, 100]
Type A	300	3000	2	[1, 100]
Type B	300	3000	5	[1, 100]
Type C	400	4000	2	[1, 100]
Type D	400	4000	5	[1, 100]
Type E	500	5000	10	[1, 100]
Type F	500	5000	20	[1, 100]
Type G	1000	10000	2	[1, 100]
Type H	1000	10000	5	[1, 100]
STS A_{135}	3015	135	2.2	[1, 1]
STS A_{243}	9801	243	1.2	[1, 1]
STS A_{405}	27270	405	0.7	[1, 1]
STS A_{729}	88452	729	0.4	[1, 1]
RAIL 507	507	63009	1.2	[1, 2]
RAIL 516	516	47311	1.3	[1, 2]
RAIL 582	582	55515	1.2	[1, 2]
RAIL 2536	2536	1081841	0.4	[1, 2]
RAIL 2586	2586	920683	0.4	[1, 2]
RAIL 4284	4284	1092610	0.2	[1, 2]
RAIL 4872	4872	968672	0.2	[1, 2]

of coefficients is very low. To the best of our knowledge, optimal solutions are not known for any of RAIL instances except for RAIL 507–582.

7.2. Effect of the neighborhood sizes

In this section, to examine the effect of neighborhood sizes, we applied our algorithm of Section 6 with restricted neighborhoods $N_r(\boldsymbol{x})$, $r = 1, 2, 3$, respectively, on types E, F, G, H and RAIL. Each instance was solved ten times using different random seeds. Results are shown in Table 2. For each instance, Min, Avg and Max are the minimum, average and maximum values, respectively, of the solutions obtained in 10 runs. Time limits *time_lim* of three algorithms with N_r for all $r = 1, 2, 3$ are set to 180 seconds for types E–H, 600 seconds for RAIL 507, 516 and 582, and 18,000 s for RAIL 2536–4872. The mark '*' indicates the minimum of the average value among the three algorithms.

From the table, we can observe that the 3-flip neighborhood is more effective than the 1- and 2-flip neighborhoods. The algorithm with N_3 obtained the best results for all instances except for three instances of type H. It also obtained the best known solutions in all of the ten runs for 19 out of 27 instances. For types E, F, G and H, the effect of neighborhood sizes is small, and the minimum values of N_2 and N_3 are the same. For this reason, it seems that types E–H are too easy to compare the effectiveness of neighborhood sizes. On the other hand, the difference in solution quality is very large between N_2 and N_3 for type RAIL instances, which are much larger than others.

7.3. Results for small instances

We tested algorithm 3-FNLS on small instances of types 4–6 and A–D, whose optimal solutions are known. Each instance was solved ten times, and the minimum, average and maximum time spent to obtain

Table 2
Comparison of different sizes of neighborhood

Instance	$NB_1(x)$			$NB_2(x)$			$NB_3(x)$		
	Min	Avg	Max	Min	Avg	Max	Min	Avg	Max
E1	29	*29.0	29	29	*29.0	29	29	*29.0	29
E2	30	30.1	30	30	30.3	31	30	*30.0	30
E3	27	*27.0	27	27	*27.0	27	27	*27.0	27
E4	28	*28.0	28	28	*28.0	28	28	*28.0	28
E5	28	*28.0	28	28	*28.0	28	28	*28.0	28
F1	14	*14.0	14	14	*14.0	14	14	*14.0	14
F2	15	*15.0	15	15	*15.0	15	15	*15.0	15
F3	14	*14.0	14	14	*14.0	14	14	*14.0	14
F4	14	*14.0	14	14	*14.0	14	14	*14.0	14
F5	13	13.2	14	13	13.2	14	13	*13.0	13
G1	176	*176.0	176	176	*176.0	176	176	*176.0	176
G2	155	155.0	155	154	154.4	155	154	*154.0	154
G3	167	167.8	169	167	167.2	168	166	*166.0	166
G4	170	170.6	172	168	168.6	170	168	*168.0	168
G5	168	*168.0	168	168	168.1	169	168	*168.0	168
H1	64	64.0	64	63	63.7	64	63	*63.0	63
H2	63	63.3	64	63	*63.0	63	63	63.3	64
H3	59	59.9	61	59	*59.2	60	59	59.9	60
H4	58	*58.0	58	58	*58.0	58	58	*58.0	58
H5	55	*55.0	55	55	*55.0	55	55	55.4	56
RAIL 507	175	175.5	176	175	175.0	175	174	*174.3	175
RAIL 516	182	182.2	183	182	*182.0	182	182	*182.0	182
RAIL 582	211	211.1	212	211	*211.0	211	211	*211.0	211
RAIL 2536	691	692.4	694	691	691.4	692	691	*691.1	692
RAIL 2586	955	957.9	962	949	950.2	952	945	*946.9	948
RAIL 4284	1075	1079.5	1083	1066	1068.9	1071	1064	*1065.7	1066
RAIL 4872	1543	1546.5	1550	1533	1535.8	1539	1528	*1531.8	1534

these optimal solutions are shown in Table 3. We can observe that the optimal solutions are obtained by 3-FNLS within a few seconds for all tested instances.

7.4. Results for STS instances

Algorithm 3-FNLS was applied to instances of type STS, whose optimal solutions are not known yet. For A_{135} and A_{243}, the best values reported in [22,25] are 103 and 198, respectively, and for A_{405} and A_{729}, the best values reported in [25] are 337 and 617, respectively. We solved each instance ten times using different random seeds, and results are shown in Table 4. For each instance, the number of runs in which the best solution is found (#Min), and the minimum (Min) and maximum (Max) of the objective values are shown. Time limits *time_lim* are set to 1800 seconds for instances A_{135} and A_{243}, and 3600 seconds for instances A_{405} and A_{729}. Notably, a solution with cost 336 was found for A_{405}, which improves on the result reported in [25].

7.5. Comparison with other algorithms

Algorithm 3-FNLS was then compared with the reported results of four existing heuristic algorithms for SCP: (1) simulated annealing by Brusco, Jacobs and Thompson (denoted BJT)

Table 3
Computational time in seconds for 3-FNLS to find optimal solutions for small instances

Instance	Opt	Time (seconds)		
		Min	Avg	Max
4.1	429	1.1	1.10	1.1
4.2	512	0.5	0.50	0.5
4.3	516	0.5	0.50	0.5
4.4	494	1.1	1.10	1.1
4.5	512	0.6	0.60	0.6
4.6	560	1.1	1.10	1.1
4.7	430	1.2	1.20	1.2
4.8	492	1.3	1.30	1.3
4.9	641	1.2	1.20	1.2
4.10	514	1.1	1.10	1.1
5.1	253	1.2	1.40	1.7
5.2	302	1.5	1.56	1.9
5.3	226	1.0	1.00	1.0
5.4	242	1.3	1.32	1.4
5.5	211	1.0	1.00	1.0
5.6	213	1.1	1.10	1.1
5.7	293	1.2	1.26	1.4
5.8	288	1.2	1.20	1.2
5.9	279	1.2	1.21	1.3
5.10	265	1.1	1.10	1.1
6.1	138	1.6	1.60	1.6
6.2	146	1.4	1.45	1.7
6.3	145	1.4	1.40	1.4
6.4	131	1.9	1.90	1.9
6.5	161	1.4	1.54	1.9
A1	253	1.9	3.45	16.6
A2	252	1.9	2.03	2.4
A3	232	1.8	2.53	3.1
A4	234	2.1	2.25	2.5
A5	236	2.5	2.60	2.9
B1	69	2.5	2.53	2.6
B2	76	2.2	2.31	2.6
B3	80	2.7	2.85	3.8
B4	79	2.3	2.43	2.7
B5	72	2.7	2.80	2.9
C1	227	2.9	3.01	3.4
C2	219	2.9	2.91	3.0
C3	243	2.5	4.75	6.8
C4	219	2.9	3.34	4.1
C5	215	2.8	2.83	3.0
D1	60	3.4	3.49	3.7
D2	66	3.3	3.30	3.3
D3	72	3.8	4.11	4.9
D4	62	3.2	3.26	3.4
D5	61	3.8	3.89	4.1

Table 4
Results for type STS

Instance	Best-known in [22,25]	#Min	Min	Max
A_{135}	103	2	103	104
A_{243}	198	8	198	203
A_{405}	337	4	336	339
A_{729}	617	1	617	646

Table 5
Comparison with other algorithms

Instance	LB	Best-known	BJT	BC	CNS	CFT	3-FNLS (10 runs)			
							Min	Avg	Max	#BorE
E1	29	29	*29	*29	–	*29	29	*29.0	29	10/10
E2	28	30	*30	*30	–	*30	30	*30.0	30	10/10
E3	27	27	*27	*27	–	*27	27	*27.0	27	10/10
E4	28	28	*28	*28	–	*28	28	*28.0	28	10/10
E5	28	28	*28	*28	–	*28	28	*28.0	28	10/10
F1	14	14	*14	*14	–	*14	14	*14.0	14	10/10
F2	15	15	*15	*15	–	*15	15	*15.0	15	10/10
F3	14	14	*14	*14	–	*14	14	*14.0	14	10/10
F4	14	14	*14	*14	–	*14	14	*14.0	14	10/10
F5	13	13	*13	*13	–	*13	13	*13.0	13	10/10
G1	165	176	*176	*176	*176	*176	176	*176.0	176	10/10
G2	147	154	155	155	155	*154	154	*154.0	154	10/10
G3	153	166	*166	*166	167	*166	166	*166.0	166	10/10
G4	154	168	*168	*168	170	*168	168	*168.0	168	10/10
G5	153	168	*168	*168	169	*168	168	*168.0	168	10/10
H1	52	63	64	64	64	*63	63	*63.0	63	10/10
H2	52	63	*63	64	64	*63	63	63.3	64	7/10
H3	48	59	*59	*59	60	*59	59	59.9	60	1/10
H4	47	58	*58	*58	59	*58	58	*58.0	58	10/10
H5	46	55	*55	*55	*55	*55	55	55.4	56	6/10

BJT: simulated annealing by Brusco, Jacobs and Thompson [9]; BC: genetic algorithm by Beasley and Chu [7]; CNS: Lagrangian-based heuristic by Ceria, Nobili and Sassano [11]; CFT: Lagrangian-based heuristic by Caprara, Fischetti and Toth [10].

[9], [4] (2) genetic algorithm by Beasley and Chu (denoted BC) [7], (3) Lagrangian-based heuristic by Ceria, Nobili and Sassano (denoted CNS) [11], and (4) Lagrangian-based heuristic by Caprara, Fischetti and Toth (denoted CFT) [10]. BJT and BC were applied to types E–H, CNS was applied to types G, H and RAIL, and CFT was applied to types E–H and RAIL. Tables 5 and 6 show the best solutions obtained by these algorithms.

Each instance was then solved ten times by 3-FNLS from different random seeds. In Tables 5 and 6, Min, Avg and Max are the minimum, average and maximum values of the solutions obtained in the ten runs. Column '#BorE' (stands for 'better or equivalent') is the number of runs in which the previous best-known solution or a better solution is found. The mark '—' means that the experimental results are not reported and the mark '*' indicates the previous best known solution. For 3-FNLS, the meaning of '*' is slightly different. An '*' is marked in column Avg and not in column Min, since column Min can

[4] The results of two algorithms, called SAHNM and SAHWM, were reported in [9], but their results were the same for all instances if we take the best of ten runs.

Table 6
Comparison with other algorithms

Instance	LB	Best-known	Time limit[a]	CNS	CFT	3-FNLS (10 runs)			
						Min	Avg	Max	#BorE
RAIL 507	174	174	30[b]		175[c]	176	177.2	178	0/10
			180	*174		174	175.1	176	1/10
			300			174	174.8	175	2/10
			600			174	174.3	175	7/10
RAIL 516	182	182	30[b]	*182	*182	184	184.7	186	0/10
			180			182	182.1	183	9/10
			300			182	182.1	183	9/10
			600			182	*182.0	182	10/10
RAIL 582	211	211	30[b]		*211	211	*211.0	211	10/10
			180	*211					
RAIL 2536	685	690[d]	1800		*691	695	702.3	716	0/10
	(prev. best known: 691)		3600[b]	692	*691	691	692.1	694	3/10
			7200			691	691.2	692	8/10
			18000			691	691.1	692	9/10
			180000		691[e]				
RAIL 2586	937	945[f]	1800	951	948	949	951.3	952	0/10
	(prev. best known: 947)		3600[b]		948[g]	947	949.2	951	2/10
			7200			946	947.4	949	6/10
			18000			945	*946.9	948	7/10
			180000		946[e]				
RAIL 4284	1054	1064[f]	1800		1069[h]	1072	1073.8	1076	0/10
	(prev. best known: 1065)		3600[b]	1070	*1065	1068	1069.5	1071	0/10
			7200			1066	1067.3	1069	0/10
			18000			1064	1065.7	1066	2/10
			180000		1064[e]				
RAIL 4872	1509	1528[f]	1800	*1534	*1534	1537	1542.0	1545	0/10
	(prev. best known: 1534)		3600[b]		*1534	1534	1536.5	1540	1/10
			7200			1532	*1533.5	1535	8/10
			18000			1528	*1531.8	1534	10/10
			180000		1532[e]				

CNS: Lagrangian-based heuristic by Ceria et al. [11]; CFT: Lagrangian-based heuristic by Caprara et al. [10].

[a] Time limit for 3-FNLS is for "each" run.

[b] The time limit roughly equivalent to (or slightly larger than) the one for the FASTER competition [10,11], which CNS and CFT took part in. That is, 3,000 seconds for RAIL 507, 516 and 582 on a PC486/33, 4 MB memory, and 10,000 seconds for RAIL 2536–4872 on an HP735, 125 MHz, 256 MB memory.

[c] The solution with cost 174 was found by CFT with an ad hoc tuning.

[d] The solution with cost 690 was found by 3-FNLS after parameter tuning.

[e] The result with a longer time limit of about 500,000 CPU seconds on HP735/125, which is provided by the authors of [10].

[f] Best-known costs 947, 1065 and 1534 in [10] (those in the parentheses) are updated to 945, 1064 and 1528, respectively, by using 3-FNLS.

[g] The solution with cost 947 was found by CFT with an ad hoc tuning.

[h] The solution with cost 1069 was found by CFT at an earlier stage, and the result with this time limit is not available.

be understood as the results after spending '$10 \times$ *time_lim*' CPU seconds and may not be fair. The mark '*' is added if the average value is better than or equal to the previous best known solution, which means the previous best known solution is newly updated by 3-FNLS, or 3-FNLS gives the best known solution in *all* of the ten runs. We also show the lower bound (denoted LB) obtained from the Lagrangian dual prob-

Table 7
Performance comparison of different computers

Machine	SPECint95	Mflop/s	Estimate
Sun Ultra SPARC II, 300 MHz	12.3		1
Sun Ultra SPARC II, 336 MHz		154	1.12
Pentium 100 MHz	3.16–3.33		0.26
SGI R4000, 100 MHz		15	0.1
IBM RS/6000 375, 62.5 MHz		26	0.19
DECstation 5000/240		5.3	0.04
PC486, 33 MHz		0.94	0.007
HP 9000 Series 700 Model 735, 125 MHz	4.04		0.33

lem solved by the subgradient method or from [10] (a better value of these two is exhibited). For instances RAIL 507 and RAIL 582, the optimal values are obtained by CPLEX6.5.

Time limit *time_lim* for each run of our algorithm is set to 180 seconds for types E–H, and various time limits are examined for RAIL instances. Time limits and computers used for other four algorithms are given as follows:

BJT 10 runs with 60 (respectively, 240) seconds for types E and F (respectively, G and H) on a PC with Pentium 100 MHz.

BC 10 runs on a Silicon Graphics Indigo R4000, 100 MHz. Time limits were not given, but the average execution times were 2500–4500 seconds.

CNS 1000 seconds for types G and H and RAIL 507 and 582, and 10,000 seconds for RAIL 2586 and 4872 on an IBM RS/6000 375, 32 MB memory. 3000 seconds for RAIL 516 on a PC486/66, 16 MB memory. 10,000 seconds for RAIL 2536 and 4284 on an HP735, 125 MHz, 256 MB memory.

CFT 5000 seconds for types E–H on DECstation 5000/240. 3000 seconds for RAIL 507, 516 and 582 on a PC486/33, 4 MB memory. 10,000 seconds for RAIL 2536–4872 on an HP735, 125 MHz, 256 MB memory.

We give in Table 7 rough comparison of machines, Sun Ultra 2 Model 2300 UltraSPARC II 300 MHz, Pentium 100 MHz, Silicon Graphics Indigo R4000 100 MHz, IBM RS/6000 375 62.5 MHz, DECstation 5000/240, PC486/33, HP735 125 MHz 256 MB memory. Table 7 shows benchmark values of SPECint95 and Mflop/s. The values of SPECint95 are taken from the WWW site of SPEC (Standard Performance Evaluation Corporation), [5] and Mflop/s are the values of LINPACK benchmark in [12]. Based on these, we give rough estimates on machine speeds in the column 'estimate', where the speed of Sun Ultra 2 is normalized to one, and a larger value means that the computer is faster. The results of CNS and CFT in Table 6 are exhibited estimating their time limits according to Table 7.

From Table 5, we can observe that our algorithm 3-FNLS obtained the best known solutions in all of ten runs for all the tested instances except for H2, H3 and H5. For H2, H3 and H5, the number of runs in which the best known solution was found was 7, 1 and 6, respectively. Among other algorithms, CFT is considered to be the best with respect to the solution quality. Compared with CFT, the time limit of 3-FNLS is similar for type E–H.

From Table 6, we can observe that the solution quality of 3-FNLS is not very good if *time_lim* is short, while algorithms CNS and CFT obtained very good solutions already in early stage. Based on these, we should conclude that algorithm CFT is the best if the time limit is short. However, when a longer time limit is allowed, #BorE becomes closer to 10 except for RAIL 4284, and 3-FNLS could improve the previous

[5] http://www.specbench.org/

best known solutions for large instances. We asked the authors of [10] to run algorithm CFT with a very long time limit, which is approximately equivalent to 180,000 seconds on our computer. The results in column Min of algorithm 3-FNLS (achieved after 18000 seconds × 10 runs) are better than or equivalent to their results though the improvement is small. This comparison may not be fair as algorithm CFT is not designed for such a long time limit and comparing computation time on different computers is not easy; however, it indicates that the new best known values are not very easy to find even with long computation time. The speed of computers is growing rapidly (e.g., a PC with Pentium 4 (3 GHz) is about 10 times faster than the Sun Ultra 2), and hence we believe that algorithm 3-FNLS is worth existing.

8. Conclusion

In this paper, we proposed a local search algorithm, which is based on the 3-flip neighborhood (3-FNLS), for the set covering problem. It contains an efficient implementation of the neighborhood search, realized by reducing the neighborhood size without sacrificing the solution quality. We also incorporate the strategic oscillation mechanism based on the adaptive control of penalty weights, and the size reduction approach by using the information from Lagrangian relaxation.

Computational results show that the 3-flip neighborhood is effective to obtain better solutions than those obtained by 1-flip and 2-flip neighborhoods, which are commonly used in existing algorithms. Comparisons with other existing heuristic algorithms revealed that 3-FNLS obtained the best values for all the tested instances, and improved the best known values for some of the large-scale RAIL instances.

Acknowledgments

The authors are grateful to Alberto Caprara for providing us with the computational results of algorithm CFT in Section 7.5. They are also grateful to anonymous referees for their valuable comments, which were used to improve this paper. This research was partially supported by Scientific Grant-in-Aid by the Ministry of Education, Culture, Sports, Science and Technology of Japan, by Informatics Research Center for Development of Knowledge Society Infrastructure (COE program of the Ministry of Education, Culture, Sports, Science and Technology, Japan) and by the Telecommunications Advancement Foundation of Japan.

Appendix A. Proof of Lemma 1

We show that $\Delta pcost(\boldsymbol{x}, \{j_1, j_2\}) \geqslant 0$ holds if $x_{j_1} = x_{j_2}$. First, we consider the case of $x_{j_1} = x_{j_2} = 1$. By the assumption in the lemma

$$\Delta pcost(\boldsymbol{x}, \{j\}) = -c_j + \sum_{i \in S_j} p_i \theta_i^{(1)}(\boldsymbol{x}) \geqslant 0 \quad \forall j \in \{j_1, j_2\}$$

holds. Then we have

$$\begin{aligned}\Delta pcost(\boldsymbol{x}, \{j_1, j_2\}) &= -c_{j_1} - c_{j_2} + \sum_{i \in S_{j_1}} p_i \theta_i^{(1)}(\boldsymbol{x}) + \sum_{i \in S_{j_2}} p_i \theta_i^{(1)}(\boldsymbol{x}) + \sum_{i \in S_{j_1} \cap S_{j_2}} p_i \theta_i^{(2)}(\boldsymbol{x}) \\ &= \Delta pcost(\boldsymbol{x}, \{j_1\}) + \Delta pcost(\boldsymbol{x}, \{j_2\}) + \sum_{i \in S_{j_1} \cap S_{j_2}} p_i \theta_i^{(2)}(\boldsymbol{x}) \geqslant 0.\end{aligned}$$

Next, we consider the case of $x_{j_1} = x_{j_2} = 0$. By the assumption

$$\Delta pcost(\boldsymbol{x}, \{j\}) = c_j - \sum_{i \in S_j} p_i \theta_i^{(0)}(\boldsymbol{x}) \geqslant 0 \quad \forall j \in \{j_1, j_2\}$$

holds. Then we have

$$\begin{aligned}
\Delta pcost(\boldsymbol{x}, \{j_1, j_2\}) &= c_{j_1} + c_{j_2} - \sum_{i \in S_{j_1} \cup S_{j_2}} p_i \theta_i^{(0)}(\boldsymbol{x}) \\
&= c_{j_1} + c_{j_2} - \sum_{i \in S_{j_1}} p_i \theta_i^{(0)}(\boldsymbol{x}) - \sum_{i \in S_{j_2}} p_i \theta_i^{(0)}(\boldsymbol{x}) + \sum_{i \in S_{j_1} \cap S_{j_2}} p_i \theta_i^{(0)}(\boldsymbol{x}) \\
&= \Delta pcost(\boldsymbol{x}, \{j_1\}) + \Delta pcost(\boldsymbol{x}, \{j_2\}) + \sum_{i \in S_{j_1} \cap S_{j_2}} p_i \theta_i^{(0)}(\boldsymbol{x}) \geqslant 0. \qquad \square
\end{aligned}$$

Appendix B. Proof of Lemma 2

We assume $x_{j_1} = 1$ and $x_{j_2} = 0$ without loss of generality. By the assumption in the lemma, we have the following inequalities:

$$\Delta pcost(\boldsymbol{x}, \{j_1\}) = -c_{j_1} + \sum_{i \in S_{j_1}} p_i \theta_i^{(1)}(\boldsymbol{x}) \geqslant 0,$$

$$\Delta pcost(\boldsymbol{x}, \{j_2\}) = c_{j_2} - \sum_{i \in S_{j_2}} p_i \theta_i^{(0)}(\boldsymbol{x}) \geqslant 0.$$

Hence we have

$$\begin{aligned}
\Delta pcost(\boldsymbol{x}, \{j_1, j_2\}) &= -c_{j_1} + c_{j_2} + \sum_{i \in S_{j_1} \setminus S_{j_2}} p_i \theta_i^{(1)}(\boldsymbol{x}) - \sum_{i \in S_{j_2}} p_i \theta_i^{(0)}(\boldsymbol{x}) \\
&= -c_{j_1} + c_{j_2} + \sum_{i \in S_{j_1}} p_i \theta_i^{(1)}(\boldsymbol{x}) - \sum_{i \in S_{j_1} \cap S_{j_2}} p_i \theta_i^{(1)}(\boldsymbol{x}) - \sum_{i \in S_{j_2}} p_i \theta_i^{(0)}(\boldsymbol{x}) \\
&= \Delta pcost(\boldsymbol{x}, \{j_1\}) + \Delta pcost(\boldsymbol{x}, \{j_2\}) - \sum_{i \in S_{j_1} \cap S_{j_2}} p_i \theta_i^{(1)}(\boldsymbol{x}).
\end{aligned}$$

Therefore $\Delta pcost(\boldsymbol{x}, \{j_1, j_2\}) < 0$ holds only if $\sum_{i \in S_{j_1} \cap S_{j_2}} p_i \theta_i^{(1)}(\boldsymbol{x}) > 0$. As $p_i > 0$ for all $i \in M$, $\sum_{i \in S_{j_1} \cap S_{j_2}} p_i \theta_i^{(1)}(\boldsymbol{x}) > 0$ is equivalent to $\{i \in S_{j_1} \cap S_{j_2} \mid \theta_i^{(1)}(\boldsymbol{x}) = 1\} \neq \emptyset$. $\square$

Appendix C. Proof of Lemma 3

We show that $\Delta pcost(\boldsymbol{x}, \{j_1, j_2, j_3\}) \geqslant 0$ holds if $x_{j_1} = x_{j_2} = x_{j_3}$. By symmetry, this will prove Lemma 3. First, we consider the case of $x_{j_1} = x_{j_2} = x_{j_3} = 1$. By the assumption in the lemma

$$\Delta pcost(\boldsymbol{x}, \{j\}) = -c_j + \sum_{i \in S_j} p_i \theta_i^{(1)}(\boldsymbol{x}) \geqslant 0 \quad \forall j \in \{j_1, j_2, j_3\}$$

holds. Then we have

$$\begin{aligned}
\Delta pcost(\boldsymbol{x}, \{j_1, j_2, j_3\}) = &-c_{j_1} - c_{j_2} - c_{j_3} + \sum_{i \in S_{j_1}} p_i \theta_i^{(1)}(\boldsymbol{x}) + \sum_{i \in S_{j_2}} p_i \theta_i^{(1)}(\boldsymbol{x}) + \sum_{i \in S_{j_3}} p_i \theta_i^{(1)}(\boldsymbol{x}) + \sum_{i \in S_{j_1} \cap S_{j_2}} p_i \theta_i^{(2)}(\boldsymbol{x}) \\
&+ \sum_{i \in S_{j_2} \cap S_{j_3}} p_i \theta_i^{(2)}(\boldsymbol{x}) + \sum_{i \in S_{j_3} \cap S_{j_1}} p_i \theta_i^{(2)}(\boldsymbol{x}) + \sum_{i \in S_{j_1} \cap S_{j_2} \cap S_{j_3}} p_i \theta_i^{(3)}(\boldsymbol{x}) \geqslant 0.
\end{aligned}$$

Next, we consider the case of $x_{j_1} = x_{j_2} = x_{j_3} = 0$. By the assumption

$$\Delta pcost(\boldsymbol{x}, \{j\}) = c_j - \sum_{i \in S_j} p_i \theta_i^{(0)}(\boldsymbol{x}) \geqslant 0 \quad \forall j \in \{j_1, j_2, j_3\}$$

holds. Then we have

$$\begin{aligned}
\Delta pcost(\boldsymbol{x}, \{j_1, j_2, j_3\}) &= c_{j_1} + c_{j_2} + c_{j_3} - \sum_{i \in S_{j_1} \cup S_{j_2} \cup S_{j_3}} p_i \theta_i^{(0)}(\boldsymbol{x}) \\
&= c_{j_1} + c_{j_2} + c_{j_3} - \sum_{i \in S_{j_1}} p_i \theta_i^{(0)}(\boldsymbol{x}) - \sum_{i \in S_{j_2}} p_i \theta_i^{(0)}(\boldsymbol{x}) - \sum_{i \in S_{j_3}} p_i \theta_i^{(0)}(\boldsymbol{x}) \\
&\quad + \sum_{i \in \{S_{j_1} \cap S_{j_2}\} \setminus S_{j_3}} p_i \theta_i^{(0)}(\boldsymbol{x}) + \sum_{i \in \{S_{j_2} \cap S_{j_3}\} \setminus S_{j_1}} p_i \theta_i^{(0)}(\boldsymbol{x}) + \sum_{i \in \{S_{j_3} \cap S_{j_1}\} \setminus S_{j_2}} p_i \theta_i^{(0)}(\boldsymbol{x}) \\
&\quad + 2 \sum_{i \in S_{j_1} \cap S_{j_2} \cap S_{j_3}} p_i \theta_i^{(0)}(\boldsymbol{x}) \geqslant 0. \qquad \square
\end{aligned}$$

Appendix D. Proof of Lemma 4

First, we consider the case of $x_{j_1} = x_{j_3} = 1$ and $x_{j_2} = 0$. By the assumption in the lemma, we have the following inequalities:

$$\Delta pcost(\boldsymbol{x}, \{j\}) = -c_j + \sum_{i \in S_j} p_i \theta_i^{(1)}(\boldsymbol{x}) \geqslant 0 \quad \forall j \in \{j_1, j_3\},$$

$$\Delta pcost(\boldsymbol{x}, \{j, j_2\}) = -c_j + c_{j_2} + \sum_{i \in S_j \setminus S_{j_2}} p_i \theta_i^{(1)}(\boldsymbol{x}) - \sum_{i \in S_{j_2}} p_i \theta_i^{(0)}(\boldsymbol{x}) \geqslant 0 \quad \forall j \in \{j_1, j_3\}.$$

Hence we have

$$\begin{aligned}
\Delta pcost(\boldsymbol{x}, \{j_1, j_2, j_3\}) &= -c_{j_1} + c_{j_2} - c_{j_3} + \sum_{i \in S_{j_1} \setminus S_{j_2}} p_i \theta_i^{(1)}(\boldsymbol{x}) + \sum_{i \in S_{j_3} \setminus S_{j_2}} p_i \theta_i^{(1)}(\boldsymbol{x}) \\
&\quad + \sum_{i \in \{S_{j_1} \cap S_{j_3}\} \setminus S_{j_2}} p_i \theta_i^{(2)}(\boldsymbol{x}) - \sum_{i \in S_{j_2}} p_i \theta_i^{(0)}(\boldsymbol{x}) \\
&= -c_{j_1} + c_{j_2} - c_{j_3} + \sum_{i \in S_{j_1}} p_i \theta_i^{(1)}(\boldsymbol{x}) - \sum_{i \in S_{j_1} \cap S_{j_2}} p_i \theta_i^{(1)}(\boldsymbol{x}) + \sum_{i \in S_{j_3} \setminus S_{j_2}} p_i \theta_i^{(1)}(\boldsymbol{x}) \\
&\quad + \sum_{i \in \{S_{j_1} \cap S_{j_3}\} \setminus S_{j_2}} p_i \theta_i^{(2)}(\boldsymbol{x}) - \sum_{i \in S_{j_2}} p_i \theta_i^{(0)}(\boldsymbol{x}) \\
&= \Delta pcost(\boldsymbol{x}, \{j_1\}) + \Delta pcost(\boldsymbol{x}, \{j_2, j_3\}) + \sum_{i \in \{S_{j_1} \cap S_{j_3}\} \setminus S_{j_2}} p_i \theta_i^{(2)}(\boldsymbol{x}) - \sum_{i \in S_{j_1} \cap S_{j_2}} p_i \theta_i^{(1)}(\boldsymbol{x}).
\end{aligned}$$

Similarly we have

$$\Delta pcost(\boldsymbol{x}, \{j_1, j_2, j_3\}) = \Delta pcost(\boldsymbol{x}, \{j_1, j_2\}) + \Delta pcost(\boldsymbol{x}, \{j_3\}) + \sum_{i \in \{S_{j_1} \cap S_{j_3}\} \setminus S_{j_2}} p_i \theta_i^{(2)}(\boldsymbol{x}) - \sum_{i \in S_{j_2} \cap S_{j_3}} p_i \theta_i^{(1)}(\boldsymbol{x}).$$

Therefore $\Delta pcost(\boldsymbol{x}, \{j_1, j_2, j_3\}) < 0$ holds only if both $\sum_{i \in S_{j_1} \cap S_{j_2}} p_i \theta_i^{(1)}(\boldsymbol{x}) > 0$ and $\sum_{i \in S_{j_2} \cap S_{j_3}} p_i \theta_i^{(1)}(\boldsymbol{x}) > 0$ hold. Clearly $\sum_{i \in S_{j_1} \cap S_{j_2}} p_i \theta_i^{(1)}(\boldsymbol{x}) > 0$ (respectively, $\sum_{i \in S_{j_2} \cap S_{j_3}} p_i \theta_i^{(1)}(\boldsymbol{x}) > 0$) is equivalent to $\{i \in S_{j_1} \cap S_{j_2} \mid \theta_i^{(1)}(\boldsymbol{x}) = 1\} \neq \emptyset$ (respectively, $\{i \in S_{j_2} \cap S_{j_3} \mid \theta_i^{(1)}(\boldsymbol{x}) = 1\} \neq \emptyset$).

Next, we consider the case of $x_{j_1} = 1$ and $x_{j_2} = x_{j_3} = 0$. By the assumption in the lemma, we have the following inequalities:

$$\Delta pcost(\boldsymbol{x}, \{j\}) = c_j - \sum_{i \in S_j} p_i \theta_i^{(0)}(\boldsymbol{x}) \geqslant 0 \quad \forall j \in \{j_2, j_3\},$$
$$\Delta pcost(\boldsymbol{x}, \{j_1, j\}) = -c_{j_1} + c_j + \sum_{i \in S_{j_1} \setminus S_j} p_i \theta_i^{(1)}(\boldsymbol{x}) - \sum_{i \in S_j} p_i \theta_i^{(0)}(\boldsymbol{x}) \geqslant 0 \quad \forall j \in \{j_2, j_3\}.$$

Hence we have

$$\begin{aligned}
\Delta pcost(\boldsymbol{x}, \{j_1, j_2, j_3\}) &= -c_{j_1} + c_{j_2} + c_{j_3} + \sum_{i \in S_{j_1} \setminus \{S_{j_2} \cup S_{j_3}\}} p_i \theta_i^{(1)}(\boldsymbol{x}) - \sum_{i \in S_{j_2} \cup S_{j_3}} p_i \theta_i^{(0)}(\boldsymbol{x}) \\
&= -c_{j_1} + c_{j_2} + c_{j_3} + \sum_{i \in S_{j_1} \setminus S_{j_3}} p_i \theta_i^{(1)}(\boldsymbol{x}) - \sum_{i \in \{S_{j_1} \cap S_{j_2}\} \setminus S_{j_3}} p_i \theta_i^{(1)}(\boldsymbol{x}) \\
&\quad - \sum_{i \in S_{j_2}} p_i \theta_i^{(0)}(\boldsymbol{x}) - \sum_{i \in S_{j_3}} p_i \theta_i^{(0)}(\boldsymbol{x}) + \sum_{i \in S_{j_2} \cap S_{j_3}} p_i \theta_i^{(0)}(\boldsymbol{x}) \\
&= \Delta pcost(\boldsymbol{x}, \{j_1, j_3\}) + \Delta pcost(\boldsymbol{x}, \{j_2\}) + \sum_{i \in S_{j_2} \cap S_{j_3}} p_i \theta_i^{(0)}(\boldsymbol{x}) - \sum_{i \in \{S_{j_1} \cap S_{j_2}\} \setminus S_{j_3}} p_i \theta_i^{(1)}(\boldsymbol{x}).
\end{aligned}$$

Similarly we have

$$\Delta pcost(\boldsymbol{x}, \{j_1, j_2, j_3\}) = \Delta pcost(\boldsymbol{x}, \{j_1, j_2\}) + \Delta pcost(\boldsymbol{x}, \{j_3\}) + \sum_{i \in S_{j_2} \cap S_{j_3}} p_i \theta_i^{(0)}(\boldsymbol{x}) - \sum_{i \in \{S_{j_1} \cap S_{j_3}\} \setminus S_{j_2}} p_i \theta_i^{(1)}(\boldsymbol{x}).$$

Therefore $\Delta pcost(\boldsymbol{x}, \{j_1, j_2, j_3\}) < 0$ holds only if both $\sum_{i \in \{S_{j_1} \cap S_{j_2}\} \setminus S_{j_3}} p_i \theta_i^{(1)}(\boldsymbol{x}) > 0$ and $\sum_{i \in \{S_{j_1} \cap S_{j_3}\} \setminus S_{j_2}} p_i \theta_i^{(1)}(\boldsymbol{x}) > 0$ hold. Clearly $\sum_{i \in \{S_{j_1} \cap S_{j_2}\} \setminus S_{j_3}} p_i \theta_i^{(1)}(\boldsymbol{x}) > 0$ (respectively, $\sum_{i \in \{S_{j_1} \cap S_{j_3}\} \setminus S_{j_2}} p_i \theta_i^{(1)}(\boldsymbol{x}) > 0$) is equivalent to $\{i \in \{S_{j_1} \cap S_{j_2}\} \setminus S_{j_3} \mid \theta_i^{(1)}(\boldsymbol{x}) = 1\} \neq \emptyset$ (respectively, $\{i \in \{S_{j_1} \cap S_{j_3}\} \setminus S_{j_2} \mid \theta_i^{(1)}(\boldsymbol{x}) = 1\} \neq \emptyset$). □

Appendix E. Step size of adjusting penalty weights

In this section, we show that the value of $\delta^+(\boldsymbol{x}^{\text{prev}})$ is chosen so that $|\{\boldsymbol{x}' \in \text{NB}_2(\boldsymbol{x}^{\text{prev}}) | pcost(\boldsymbol{x}') < pcost(\boldsymbol{x}^{\text{prev}})\}| \geqslant 1$ holds for the new values of p_i, and the value of $\delta^-(\boldsymbol{x}^{\boldsymbol{p}^{\text{prev}}})$ is chosen so that $|\{j \in N \mid \Delta pcost(\boldsymbol{x}^{\boldsymbol{p}^{\text{prev}}}, \{j\}) < 0, x_j^{\text{prev}} = 1\}| \geqslant k'$ holds for the new values of p_i, if $\delta_{k'}^-(\boldsymbol{x}^{\text{prev}}) < 1 - \varepsilon^-$ holds. To clarify the argument, we denote the change in *pcost* under the vector $\boldsymbol{p}$ of penalty weight as $\Delta pcost(\boldsymbol{x}^{\text{prev}}, \{j\}, \boldsymbol{p})$.

Case 1 (Condition (2). holds and the penalty weights are increased by (3)): For a parameter δ^+ (> 0), let $p_i' = p_i(1 + \theta_i^{(0)}(\boldsymbol{x}^{\text{prev}})\delta^+)$. First we consider the 1-flip neighborhood. For $j \in N$ such that $x_j^{\text{prev}} = 0$, we have

$$\begin{aligned}
\Delta pcost(\boldsymbol{x}^{\text{ prev}}, \{j\}, \boldsymbol{p}') &= c_j - \sum_{i \in S_j} p_i' \theta_i^{(0)}(\boldsymbol{x}^{\text{prev}}) \\
&= c_j - \sum_{i \in S_j} p_i \theta_i^{(0)}(\boldsymbol{x}^{\text{prev}}) - \delta^+ \sum_{i \in S_j} p_i \theta_i^{(0)}(\boldsymbol{x}^{\text{prev}}) \\
&= \Delta pcost(\boldsymbol{x}^{\text{prev}}, \{j\}, \boldsymbol{p}) - \delta^+ \sum_{i \in S_j} p_i \theta_i^{(0)}(\boldsymbol{x}^{\text{prev}}).
\end{aligned}$$

Even with the rules in Section 3.3, $\Delta\, pcost(\boldsymbol{x}^{\mathrm{prev}}, \{j\}, \boldsymbol{p}) \geqslant 0$ holds for all $j \in N$ with $x_j^{\mathrm{prev}} = 0$ by condition (2). Hence we have

$$\Delta pcost(\boldsymbol{x}^{\mathrm{prev}}, \{j\}, \boldsymbol{p}') < 0 \Longleftrightarrow \delta^+ > \frac{\Delta pcost(\boldsymbol{x}^{\mathrm{prev}}, \{j\}, \boldsymbol{p})}{\sum_{i \in S_j} p_i \theta_i^{(0)}(\boldsymbol{x}^{\mathrm{prev}})} \quad \text{and} \quad \sum_{i \in S_j} p_i \theta_i^{(0)}(\boldsymbol{x}^{\mathrm{prev}}) > 0.$$

By the definition, $\delta^+_{\mathrm{NB}_1}(\boldsymbol{x}^{\mathrm{prev}})$ is the infimum of such δ^+. Next we consider the 2-flip neighborhood. For j_1 and j_2 with $x_{j_1}^{\mathrm{prev}} = 1$ and $x_{j_2}^{\mathrm{prev}} = 0$, we have

$$\begin{aligned}
\Delta pcost(\boldsymbol{x}^{\mathrm{prev}}, \{j_1, j_2\}, \boldsymbol{p}') &= -c_{j_1} + c_{j_2} - \sum_{i \in S_{j_2}} p_i' \theta_i^{(0)}(\boldsymbol{x}^{\mathrm{prev}}) + \sum_{i \in S_{j_1} \setminus S_{j_2}} p_i' \theta_i^{(1)}(\boldsymbol{x}^{\mathrm{prev}}) \\
&= -c_{j_1} + c_{j_2} - \sum_{i \in S_{j_2}} p_i \theta_i^{(0)}(\boldsymbol{x}^{\mathrm{prev}}) + \sum_{i \in S_{j_1} \setminus S_{j_2}} p_i \theta_i^{(1)}(\boldsymbol{x}^{\mathrm{prev}}) - \delta^+ \sum_{i \in S_{j_2}} p_i \theta_i^{(0)}(\boldsymbol{x}^{\mathrm{prev}}) \\
&= \Delta pcost(\boldsymbol{x}^{\mathrm{prev}}, \{j_1, j_2\}, \boldsymbol{p}) - \delta^+ \sum_{i \in S_{j_2}} p_i \theta_i^{(0)}(\boldsymbol{x}^{\mathrm{prev}}).
\end{aligned}$$

Therefore

$$\Delta pcost(\boldsymbol{x}^{\mathrm{prev}}, \{j_1, j_2\}, \boldsymbol{p}') < 0 \Longleftrightarrow \sum_{i \in S_{j_2}} p_i \theta_i^{(0)}(\boldsymbol{x}^{\mathrm{prev}}) > 0 \quad \text{and} \quad \delta^+ > \frac{\Delta pcost(\boldsymbol{x}^{\mathrm{prev}}, \{j_1, j_2\}, \boldsymbol{p})}{\sum_{i \in S_{j_2}} p_i \theta_i^{(0)}(\boldsymbol{x}^{\mathrm{prev}})}$$

holds provided $\Delta pcost(\boldsymbol{x}^{\mathrm{prev}}, \{j_1, j_2\}, \boldsymbol{p}) \geqslant 0$. By the definition, $\delta^+_{\mathrm{NB}_2}(\boldsymbol{x}^{\mathrm{prev}})$ is the infimum of such δ^+.

Case 2 (Condition (2) does not hold and the penalty weights are decreased by (4)): For a parameter δ^- (> 0), let $p_i' = p_i(1 - \delta^-)$. For $j \in N$ such that $x_j^{\mathrm{prev}} = 1$, we have

$$\begin{aligned}
\Delta pcost(\boldsymbol{x}^{\mathrm{prev}}, \{j\}, \boldsymbol{p}') &= -c_j + \sum_{i \in S_j} p_i' \theta_i^{(1)}(\boldsymbol{x}^{\mathrm{prev}}) = -c_j + \sum_{i \in S_j} p_i \theta_i^{(1)}(\boldsymbol{x}^{\mathrm{prev}}) - \delta^- \sum_{i \in S_j} p_i \theta_i^{(1)}(\boldsymbol{x}^{\mathrm{prev}}) \\
&= \Delta pcost(\boldsymbol{x}^{\mathrm{prev}}, \{j\}, \boldsymbol{p}) - \delta^- \sum_{i \in S_j} p_i \theta_i^{(1)}(\boldsymbol{x}^{\mathrm{prev}}).
\end{aligned}$$

As $\Delta pcost(\boldsymbol{x}^{\mathrm{prev}}, \{j\}, \boldsymbol{p}) \geqslant 0$ holds for all $j \in N$ with $x_j^{\mathrm{prev}} = 1$, we have $\sum_{i \in S_j} p_i \theta_i^{(1)}(\boldsymbol{x}^{\mathrm{prev}}) > 0$, and have

$$\Delta pcost(\boldsymbol{x}^{\mathrm{prev}}, \{j\}, \boldsymbol{p}') < 0 \Longleftrightarrow \delta^- > \frac{\Delta pcost(\boldsymbol{x}^{\mathrm{prev}}, \{j\}, \boldsymbol{p})}{\sum_{i \in S_j} p_i \theta_i^{(1)}(\boldsymbol{x}^{\mathrm{prev}})}.$$

The parameters β^+, β^-, ε^+ and ε^- are used to ensure the strict inequality except for the rule in (5). In case $\delta^-_{k'}(\boldsymbol{x}^{\mathrm{prev}}) \geqslant 1 - \varepsilon^-$ holds (which will not happen in normal situations), $\delta^-(\boldsymbol{x}^{\mathrm{prev}})$ is enforced to be $1 - \varepsilon^-$ and the above property does not hold.

References

[1] U. Aickelin, An indirect genetic algorithm for set covering problems, Journal of the Operational Research Society 53 (2002) 1118–1126.

[2] E. Balas, M.C. Carrera, A dynamic subgradient-based branch and bound procedure for set covering, Operations Research 44 (1996) 875–890.

[3] E. Balas, A. Ho, Set covering algorithms using cutting planes, heuristics, and subgradient optimization: A computational study, Mathematical Programming Study 12 (1980) 37–60.
[4] J.E. Beasley, An algorithm for set covering problems, European Journal of Operational Research 31 (1990) 85–93.
[5] J.E. Beasley, A Lagrangian heuristic for set-covering problems, Naval Research Logistics 37 (1990) 151–164.
[6] J.E. Beasley, K. Jornsten, Enhancing an algorithm for set covering problems, European Journal of Operational Research 58 (1992) 293–300.
[7] J.E. Beasley, P.C. Chu, A genetic algorithm for set covering problem, European Journal of Operational Research 94 (1996) 392–404.
[8] E. Boros, P.L. Hammer, T. Ibaraki, A. Kogan, Logical analysis of numerical data, Mathematical Programming 79 (1997) 163–190.
[9] M.J. Brusco, L.W. Jacobs, G.M. Thompson, A morphing procedure to supplement a simulated annealing heuristic for cost- and coverage-correlated set-covering problems, Annals of Operations Research 86 (1999) 611–627.
[10] A. Caprara, M. Fischetti, P. Toth, A heuristic method for the set covering problem, Operations Research 47 (1999) 730–743.
[11] S. Ceria, P. Nobili, A. Sassano, A Lagrangian-based heuristic for large-scale set covering problems, Mathematical Programming 81 (1998) 215–228.
[12] J.J. Dongarra, Performance of Various Computers Using Standard Linear Equations Software, Technical Report No. CS-89–85, Computer Science Department, University of Tennessee, July 1999. Available from www.netlib.org/benchmark/performance.ps).
[13] A. Feo, M.G.C. Resende, A probabilistic heuristic for a computationally difficult set covering problem, Operations Research Letters 8 (1989) 67–71.
[14] M.L. Fisher, The Lagrangian relaxation method for solving integer programming problems, Management Science 27 (1981) 1–18.
[15] M.L. Fisher, P. Kedia, Optimal solutions of set covering/partitioning problems using dual heuristics, Management Science 36 (1990) 674–688.
[16] F. Glover, Tabu search–Part I, ORSA Journal on Computing 1 (1989) 190–206.
[17] S. Haddadi, Simple Lagrangian heuristic for the set covering problem, European Journal of Operational Research 97 (1997) 200–204.
[18] M. Hall Jr., Combinatorial Theory, Blaisdell Company, Waltham, MA, 1967.
[19] S. Hanafi, A. Freville, An efficient tabu search approach for the 0–1 multidimensional knapsack problem, European Journal of Operational Research 106 (1998) 659–675.
[20] M. Held, R.M. Karp, The traveling salesman problem and minimum spanning trees: Part II, Mathematical Programming 1 (1971) 6–25.
[21] L.W. Jacobs, M.J. Brusco, A local-search heuristic for large set-covering problems, Naval Research Logistics 42 (1995) 1129–1140.
[22] M.A. Odijk, H. van Maaron, Improved solutions to Steiner triple covering problem, Information Processing Letters 65 (1998) 67–69.
[23] B.M. Smith, IMPACS–A bus crew scheduling system using integer programming, Mathematical Programming 42 (1988) 181–187.
[24] F.J. Vasko, G.R. Wilson, Using a facility location algorithm to solve large set covering problems, Operations Research Letters 3 (1984) 85–90.
[25] M. Yagiura, T. Ibaraki, Efficient 2 and 3-flip neighborhood search algorithms for the MAX SAT: Experimental evaluation, Journal of Heuristics 7 (2001) 423–442.