Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence

Exploiting Variable Associations to Configure Efficient Local Search in Large-Scale Set Partitioning Problems

Shunji Umetani
Graduate School of Information Science and Technology, Osaka University
2-1 Yamadaoka, Suita, Osaka, Japan
umetani@ist.osaka-u.ac.jp

Abstract

We present a data mining approach for reducing the search space of local search algorithms in large-scale set partitioning problems (SPPs). We construct a k-nearest neighbor graph by extracting variable associations from the instance to be solved, in order to identify promising pairs of flipping variables in the large neighborhood search. We incorporate the search space reduction technique into a 2-flip neighborhood local search algorithm with an efficient incremental evaluation of solutions and an adaptive control of penalty weights. We also develop a 4-flip neighborhood local search algorithm that flips four variables alternately along 4-paths or 4-cycles in the k-nearest neighbor graph. According to computational comparison with the latest solvers, our algorithm performs effectively for large-scale SPP instances with up to 2.57 million variables.

1 Introduction

Automated methods for designing algorithms called autonomous search (Hamadi, Monfroy, and Subion 2011) have recently emerged which improve both the efficiency and applicability of algorithms by extracting useful features from the instance to be solved. It would certainly be difficult to extract the useful features for improving the efficiency of algorithms from the general form of constraint satisfaction problems (CSPs) and mixed integer programming problems (MIPs). However, we note that most of the instances arising from real-world applications have representative features individually. Many MIP benchmark instances contain many special constraints that represent their combinatorial structures (Achterberg, Koch, and Martin 2006; Koch et al. 2011). We also note that it is not necessary to configure all program parameters of algorithms in advance, and instead it is possible to make some of them configurable at runtime. Autonomous search is an automated system that configures an algorithm for the instance to be solved by combining various components and tuning program parameters at runtime.

A major direction of autonomous search as found in the literature is automated algorithm configuration based on supervised learning approaches. This is commonly formulated to optimize the empirical performance of a highly parameterized algorithm for a set of training instances (Adenso-Díaz and Laguna 2006; Balaprakash, Birattari, and Stützle 2007; Hutter, Hoos, and Leyton-Brown 2009; KhudaBukhsh et al. 2009; Kadioglu et al. 2010; Crawford et al. 2013; Monfroy et al. 2013). A big advantage of this is that automated configuration approaches require no domain knowledge or human effort to tackle a new domain. However, an important drawback of these approaches is that they require many training instances and much computation time for offline learning. For example, it has been reported that it took 28 CPU days to learn 300 training instances of set partitioning problems (SPPs) with up to 50 constraints and 2000 variables (Kadioglu, Malitsky, and Sellmann 2012). This is particularly critical when solving large-scale combinatorial problems in a limited period of time.

In this paper, we consider how to extract useful features from the instance to be solved with the aim to reduce the search space of local search algorithms for large-scale SPP instances.

In designing local search algorithms for large-scale combinatorial problems, improvements in computational efficiency become more effective than improvements in sophisticated search strategy as the instance size increases. The quality of locally optimal solutions typically improves if a larger neighborhood is used. However, the computation time to search the neighborhood also increases exponentially. To overcome this, ways to efficiently implement neighborhood search have been studied extensively, and can be broadly classified into three types: (i) reducing the number of candidates in the neighborhood (Pesant and Gendreau 1999; Yagiura and Ibaraki 1999; Shaw 2002; Yagiura, Kishida, and Ibaraki 2006), (ii) evaluating solutions by incremental computation (Yagiura and Ibaraki 1999; Michel and Van Hentenryck 2000; Voudouris et al. 2001; Van Hentenryck and Michel 2005), and (iii) reducing the number of variables to be considered by using linear programming (LP) and/or Lagrangian relaxation (Ceria, Nobili, and Sassano 1998; Caprara, Fischetti, and Toth 1999; Yagiura, Kishida, and Ibaraki 2006; Umetani, Arakawa, and Yagiura 2013).

As an alternative, we have developed a data mining approach for reducing the search space of local search algorithms. I.e., we construct a k-nearest neighbor graph by ex-

tracting variable associations from the instance to be solved in order to identify promising pairs of flipping variables in the large neighborhood search. We incorporate the search space reduction technique into a 2-flip local search algorithm which offers efficient incremental evaluation of solutions and adaptive control of penalty weights. We also develop a 4-flip neighborhood local search algorithm that flips four variables alternately along 4-paths or 4-cycles in the k-nearest neighbor graph. Comparison of computations against the latest solvers shows that our algorithm performs effectively for large-scale SPP instances with up to 2.57 million variables.

2 Set Partitioning Problem

The SPP is a representative combinatorial optimization problem that has many real-world applications, such as crew scheduling and vehicle routing. We are given a ground set of m elements $i \in M = \{1, \ldots, m\}$, n subsets $S_j \subseteq M$ ($|S_j| \geq 1$) and their costs $c_j \in \mathbb{R}$ for $j \in N = \{1, \ldots, n\}$. We say that $X \subseteq N$ is a partition of M if $\bigcup_{j \in X} S_j = M$ and $S_{j_1} \cap S_{j_2} = \emptyset$ hold for all $j_1, j_2 \in X$. The goal of the SPP is to find a minimum cost partition X of M. The SPP is formulated as a 0-1 integer programming (0-1IP) problem as follows:

$$\begin{aligned} \text{min.} \quad & \sum_{j \in N} c_j x_j \\ \text{s.t.} \quad & \sum_{j \in N} a_{ij} x_j = 1, \quad i \in M, \\ & x_j \in \{0, 1\}, \qquad j \in N, \end{aligned} \tag{1}$$

where $a_{ij} = 1$ if $i \in S_j$ holds and $a_{ij} = 0$ otherwise, and $x_j = 1$ if $j \in X$ and $x_j = 0$ otherwise, respectively. That is, a column vector $\boldsymbol{a}_j = (a_{1j}, \ldots, a_{mj})^T$ of matrix (a_{ij}) represents the corresponding subset S_j by $S_j = \{i \in M \mid a_{ij} = 1\}$, and the vector $\boldsymbol{x}$ also represents the corresponding partition X by $X = \{j \in N \mid x_j = 1\}$.

Although the SPP is known to be NP-hard in the strong sense, several efficient exact and heuristic algorithms for large-scale SPP instances have been developed in the literature (Atamtürk, Nemhauser, and Savelsbergh 1995; Wedelin 1995; Borndörfer 1998; Chu and Beasley 1998; Barahona and Anbil 2000; Linderoth, Lee, and Savelsbergh 2001; Boschetti, Mingozzi, and Ricciardelli 2008; Bastert, Hummel, and de Vries 2010). Many of them are based on the variable fixing techniques that reduce the search space to be explored by using lower bounds of the optimal values obtained from linear programming (LP) and/or Lagrangian relaxation. However, many large-scale SPP instances still remain unsolved because there is little hope of closing the large gap between the lower and upper bounds of the optimal values.

3 2-flip Neighborhood Local Search

The local search (LS) starts from an initial solution $\boldsymbol{x}$ and then repeatedly replaces $\boldsymbol{x}$ with a better solution $\boldsymbol{x}'$ in its neighborhood $\mathrm{NB}(\boldsymbol{x})$ until no better solution is found in $\mathrm{NB}(\boldsymbol{x})$. For some positive integer r, let the r-flip neighborhood $\mathrm{NB}_r(\boldsymbol{x})$ be the set of solutions obtainable by flipping at most r variables of $\boldsymbol{x}$. We first develop a 2-flip neighborhood local search (2-FNLS) algorithm as a basic component of our algorithm. In order to improve efficiency, the 2-FNLS first searches $\mathrm{NB}_1(\boldsymbol{x})$, and then searches $\mathrm{NB}_2(\boldsymbol{x}) \setminus \mathrm{NB}_1(\boldsymbol{x})$ only if $\boldsymbol{x}$ is locally optimal with respect to $\mathrm{NB}_1(\boldsymbol{x})$.

The SPP is NP-hard, and the (supposedly) simpler problem of judging the existence of a feasible solution is NP-complete, since the satisfiability (SAT) problem can be reduced to this problem. We accordingly consider the following formulation of SPP that allows violations of the partitioning constraints and introduce over and under penalty functions with penalty weight vectors $\boldsymbol{w}^+, \boldsymbol{w}^- \in \mathbb{R}^m_+$:

$$\begin{aligned} \text{min.} \quad & z(\boldsymbol{x}) = \sum_{j \in N} c_j x_j + \sum_{i \in M} w_i^+ y_i^+ + \sum_{i \in M} w_i^- y_i^- \\ \text{s.t.} \quad & \sum_{j \in N} a_{ij} x_j - y_i^+ + y_i^- = 1, \quad i \in M, \\ & x_j \in \{0, 1\}, \qquad j \in N, \\ & y_i^+, y_i^- \geq 0, \qquad i \in M. \end{aligned} \tag{2}$$

For a given $\boldsymbol{x} \in \{0, 1\}^n$, we can easily compute optimal $y_i^+(\boldsymbol{x}) = \max\{\sum_{j \in N} a_{ij} x_j - 1, 0\}$ and $y_i^-(\boldsymbol{x}) = \max\{1 - \sum_{j \in N} a_{ij} x_j, 0\}$. We note that when $\boldsymbol{y}^{+^*} = \boldsymbol{y}^{-^*} = \boldsymbol{0}$ holds for an optimal solution $(\boldsymbol{x}^*, \boldsymbol{y}^{+^*}, \boldsymbol{y}^{-^*})$ under the soft partitioning constraints, $\boldsymbol{x}^*$ is also optimal under the original (hard) partitioning constraints. Moreover, for an optimal solution $\boldsymbol{x}^*$ under the hard partitioning constraints, $(\boldsymbol{x}^*, \boldsymbol{0}, \boldsymbol{0})$ is also optimal with respect to the soft partitioning constraints if the values of penalty weights w_i^+, w_i^- ($i \in M$) are sufficiently large.

Since the region searched in a single application of LS is limited, LS is usually applied many times. When a locally optimal solution is found, a standard strategy is to update the penalty weights and to resume LS from the obtained locally optimal solution. We accordingly evaluate solutions with an alternative evaluation function $\tilde{z}(\boldsymbol{x})$, where the original penalty weight vectors $\boldsymbol{w}^+$, $\boldsymbol{w}^-$ are replaced with $\widetilde{\boldsymbol{w}}^+, \widetilde{\boldsymbol{w}}^-$, which are adaptively controlled in the search (See details in Section 5).

We first describe how 2-FNLS is used to search $\mathrm{NB}_1(\boldsymbol{x})$, which is called the 1-flip neighborhood. Let $\Delta\tilde{z}_j^{\uparrow}(\boldsymbol{x})$ and $\Delta\tilde{z}_j^{\downarrow}(\boldsymbol{x})$ denote the increases in $\tilde{z}(\boldsymbol{x})$ due to flipping $x_j = 0 \to 1$ and $x_j = 1 \to 0$, respectively. 2-FNLS first searches for an improved solution by flipping $x_j = 0 \to 1$ for $j \in N \setminus X$. If an improved solution is found, it chooses j with the minimum $\Delta\tilde{z}_j^{\uparrow}(\boldsymbol{x})$; otherwise, it searches for an improved solution by flipping $x_j = 1 \to 0$ for $j \in X$.

We next describe how 2-FNLS is used to search $\mathrm{NB}_2(\boldsymbol{x}) \setminus \mathrm{NB}_1(\boldsymbol{x})$, which is called the 2-flip neighborhood. We derive conditions that reduce the number of candidates in $\mathrm{NB}_2(\boldsymbol{x}) \setminus \mathrm{NB}_1(\boldsymbol{x})$ without sacrificing the solution quality by expanding the results as shown in (Yagiura, Kishida, and Ibaraki 2006). Let $\Delta\tilde{z}_{j_1,j_2}(\boldsymbol{x})$ denote the increase in $\tilde{z}(\boldsymbol{x})$ due to simultaneously flipping the values of x_{j_1} and x_{j_2}.

Lemma 1 *Suppose that a solution $\boldsymbol{x}$ is locally optimal with respect to $\mathrm{NB}_1(\boldsymbol{x})$. Then $\Delta\tilde{z}_{j_1,j_2}(\boldsymbol{x}) < 0$ holds, only if $x_{j_1} \neq x_{j_2}$.*

Based on this lemma, we consider only the set of solutions obtainable by simultaneously flipping $x_{j_1} = 1 \to 0$ and $x_{j_2} = 0 \to 1$. We now define

$$\Delta\tilde{z}_{j_1,j_2}(\boldsymbol{x}) = \Delta\tilde{z}^{\downarrow}_{j_1}(\boldsymbol{x}) + \Delta\tilde{z}^{\uparrow}_{j_2}(\boldsymbol{x}) - \sum_{i\in\bar{S}(\boldsymbol{x})} \left(\widetilde{w}^{+}_i + \widetilde{w}^{-}_i\right), \tag{3}$$

where $\bar{S}(\boldsymbol{x}) = \{i \in S_{j_1} \cap S_{j_2} \mid s_i(\boldsymbol{x}) = 1\}$.

Lemma 2 *Suppose that a solution $\boldsymbol{x}$ is locally optimal with respect to $\mathrm{NB}_1(\boldsymbol{x})$, $x_{j_1} = 1$ and $x_{j_2} = 0$. Then $\Delta\tilde{z}_{j_1,j_2}(\boldsymbol{x}) < 0$ holds, only if $\bar{S}(\boldsymbol{x}) \neq \emptyset$.*

By Lemmas 1 and 2, the 2-flip neighborhood can be restricted to the set of solutions satisfying $x_{j_1} \neq x_{j_2}$ and $\bar{S}(\boldsymbol{x}) \neq \emptyset$. However, it may not be possible to search this set efficiently without first extracting it. We thus construct a neighbor list that stores promising pairs of variables x_{j_1} and x_{j_2} for efficiency (See details in Section 4).

To increase the efficiency of 2-FNLS, we decompose the neighborhood $\mathrm{NB}_2(\boldsymbol{x})$ into a number of sub-neighborhoods. Let $\mathrm{NB}_2^{(j_1)}$ denote the subset of $\mathrm{NB}_2(\boldsymbol{x})$ obtainable by flipping $x_{j_1} = 1 \to 0$. 2-FNLS searches $\mathrm{NB}_2^{(j_1)}(\boldsymbol{x})$ for each $j_1 \in X$ in the ascending order of $\Delta\tilde{z}^{\downarrow}_{j_1}(\boldsymbol{x})$. If an improved solution is found, it chooses a pair j_1 and j_2 with the minimum $\Delta\tilde{z}_{j_1,j_2}(\boldsymbol{x})$ among those in $\mathrm{NB}_2^{(j_1)}(\boldsymbol{x})$. 2-FNLS is formally described as follows.

Algorithm 2-FNLS$(\boldsymbol{x}, \widetilde{\boldsymbol{w}}^{+}, \widetilde{\boldsymbol{w}}^{-})$

Input: A solution $\boldsymbol{x}$ and penalty weight vectors $\widetilde{\boldsymbol{w}}^{+}$ and $\widetilde{\boldsymbol{w}}^{-}$.

Output: A solution $\boldsymbol{x}$.

Step 1: If $I^{\uparrow}_1(\boldsymbol{x}) = \{j \in N \setminus X \mid \Delta\tilde{z}^{\uparrow}_j(\boldsymbol{x}) < 0\} \neq \emptyset$ holds, choose $j \in I^{\uparrow}_1(\boldsymbol{x})$ with the minimum $\Delta\tilde{z}^{\uparrow}_j(\boldsymbol{x})$, set $x_j \leftarrow 1$ and return to Step 1.

Step 2: If $I^{\downarrow}_1(\boldsymbol{x}) = \{j \in X \mid \Delta\tilde{z}^{\downarrow}_j(\boldsymbol{x}) < 0\} \neq \emptyset$ holds, choose $j \in I^{\downarrow}_1(\boldsymbol{x})$ with the minimum $\Delta\tilde{z}^{\downarrow}_j(\boldsymbol{x})$, set $x_j \leftarrow 0$ and return to Step 2.

Step 3: For each $j_1 \in X$ in the ascending order of $\Delta\tilde{z}^{\downarrow}_{j_1}(\boldsymbol{x})$, if $I_2(\boldsymbol{x}) = \{j_2 \in N \setminus X \mid \Delta\tilde{z}_{j_1,j_2}(\boldsymbol{x}) < 0\} \neq \emptyset$ holds, choose $j_2 \in I_2(\boldsymbol{x})$ with the minimum $\Delta\tilde{z}_{j_1,j_2}(\boldsymbol{x})$ and set $x_{j_1} \leftarrow 0$ and $x_{j_2} \leftarrow 1$. If the current solution $\boldsymbol{x}$ has been updated at least once in Step 3, return to Step 1; otherwise output $\boldsymbol{x}$ and exit.

If implemented naively, 2-FNLS requires $\mathrm{O}(\sigma)$ time to compute the value of the evaluation function $\tilde{z}(\boldsymbol{x})$ for the current solution $\boldsymbol{x}$, where $\sigma = \sum_{i\in M}\sum_{j\in N} a_{ij}$ denote the number of non-zero elements in the constraint matrix. This computation is quite expensive if we evaluate the neighbor solutions of the current solution $\boldsymbol{x}$ independently. To overcome this, we develop an efficient method for incrementally evaluating $\Delta\tilde{z}^{\uparrow}_j(\boldsymbol{x})$ and $\Delta\tilde{z}^{\downarrow}_j(\boldsymbol{x})$ in $\mathrm{O}(1)$ time by keeping auxiliary data in memory. By using this, 2-FNLS is also able to evaluate $\Delta\tilde{z}_{j_1,j_2}(\boldsymbol{x})$ in $\mathrm{O}(|S_j|)$ time by using (3).

4 Exploiting Variable Associations

We recall that the quality of locally optimal solutions improves if a larger neighborhood is used. However, the computation time to search the neighborhood $\mathrm{NB}_r(\boldsymbol{x})$ also increases exponentially with r, since $|\mathrm{NB}_r(\boldsymbol{x})| = \mathrm{O}(n^r)$ holds substantially. A large amount of computational time is thus needed in practice in order to scan all candidates in $\mathrm{NB}_2(\boldsymbol{x})$ for large-scale instances with millions of variables. To overcome this, we develop a data mining approach that identifies promising pairs of flipping variables in $\mathrm{NB}_2(\boldsymbol{x})$ by extracting variable associations from the instance to be solved using only a small amount of computation time.

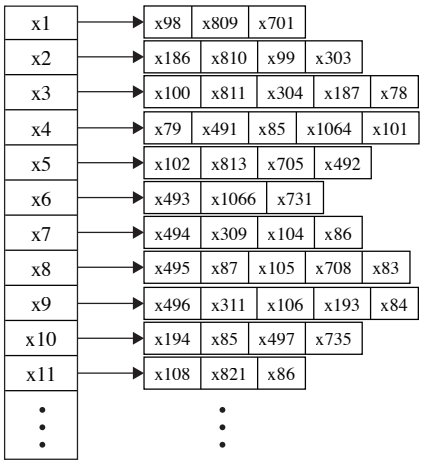

Figure 1: An example of the neighbor list

Based on Lemmas 1 and 2, the 2-flip neighborhood can be restricted to the set of solutions satisfying $x_{j_1} \neq x_{j_2}$ and $\bar{S}(\boldsymbol{x}) \neq \emptyset$. We further observe from (3) that it is favorable to select pairs of flipping variables x_{j_1} and x_{j_2} with larger $|S_{j_1} \cap S_{j_2}|$ for attaining $\Delta\tilde{z}_{j_1,j_2}(\boldsymbol{x}) < 0$. Based on this observation, we keep limited pairs of variables x_{j_1} and x_{j_2} with large $|S_{j_1} \cap S_{j_2}|$ in memory, called the neighbor list (Figure 1). We note that $|S_{j_1} \cap S_{j_2}|$ represents a kind of similarity between subsets S_{j_1} and S_{j_2} (or column vectors $\boldsymbol{a}_{j_1}$ and $\boldsymbol{a}_{j_2}$ of matrix (a_{ij})) and we keep the k-nearest neighbors for each subset S_j ($j \in N$) in the neighbor list.

For each variable x_{j_1} ($j_1 \in N$), we first enumerate x_{j_2} ($j_2 \in N$) satisfying $j_2 \neq j_1$ and $S_{j_1} \cap S_{j_2} \neq \emptyset$ to generate the set $L[j_1]$, and store the variables x_{j_2} ($j_2 \in L[j_1]$) with the top 10% largest $|S_{j_1} \cap S_{j_2}|$ in the j_1th row of the neighbor list. To be precise, we sort the variables x_{j_2} ($j_2 \in L[j_1]$) in the descending order of $|S_{j_1} \cap S_{j_2}|$ and store the first $\max\{\alpha|L[j_1]|, |M|\}$ variables in this order, where α is a program parameter that is set to 0.1. Let $L'[j_1]$ be the set of variables x_{j_2} stored in the j_1th row of the neighbor list. We then reduce the number of candidates in $\mathrm{NB}_2(\boldsymbol{x})$ by restricting pairs of flipping variables x_{j_1} and x_{j_2} to those in the neighbor list $j_1 \in X$ and $j_2 \in (N \setminus X) \cap L'[j_1]$.

We note that it is still expensive to construct the whole neighbor list for large-scale instances with millions of vari-

ables. To overcome this, we develop a lazy construction algorithm for the neighbor list. That is, 2-FNLS starts from an empty neighbor list and generates the j_1th row of the neighbor list $L'[j_1]$ only when 2-FNLS searches $\mathrm{NB}_2^{(j_1)}(\boldsymbol{x})$ for the first time.

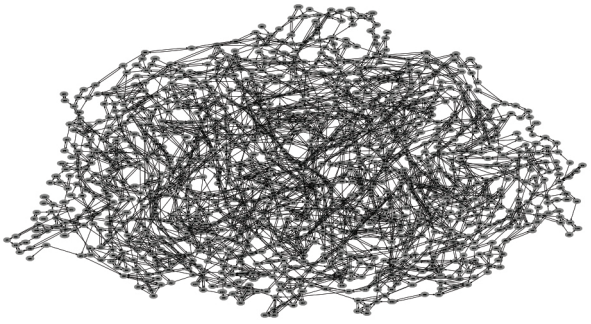

Figure 2: An example of the k-nearest neighbor graph

We can treat the neighbor list as an adjacency-list representation of a directed graph, and illustrate associations between variables by the corresponding directed graph called the k-nearest neighbor graph (Figure 2). Using the k-nearest neighbor graph, we extend 2-FNLS to search a set of promising neighbor solutions in $\mathrm{NB}_4(\boldsymbol{x})$. For each variable x_{j_1} ($j_1 \in X$), we keep $j_2 \in (N \setminus X) \cap L'[j_1]$ with the minimum $\Delta\tilde{z}_{j_1,j_2}(\boldsymbol{x})$ in memory as $\pi(j_1)$. The extended 2-FNLS, called the 4-flip neighborhood search (4-FNLS) algorithm, then searches for an improved solution by flipping $x_{j_1} = 1 \to 0$, $x_{\pi(j_1)} = 0 \to 1$, $x_{j_3} = 1 \to 0$ and $x_{\pi(j_3)} = 0 \to 1$ for $j_1 \in X$ and $j_3 \in X \cap L'[\pi(j_1)]$ satisfying $j_1 \neq j_3$ and $\pi(j_1) \neq \pi(j_3)$, i.e., flipping the values of four variables alternately along 4-paths or 4-cycles in the k-nearest neighbor graph. Let $\Delta\tilde{z}_{j_1,j_2,j_3,j_4}(\boldsymbol{x})$ denote the increase in $\tilde{z}(\boldsymbol{x})$ due to simultaneously flipping $x_{j_1} = 1 \to 0$, $x_{j_2} = 0 \to 1$, $x_{j_3} = 1 \to 0$ and $x_{j_4} = 0 \to 1$. 4-FNLS computes $\Delta\tilde{z}_{j_1,j_2,j_3,j_4}(\boldsymbol{x})$ in $\mathrm{O}(|S_j|)$ time by applying the incremental evaluation alternately. 4-FNLS is formally described by replacing the part of algorithm 2-FNLS after Step 2 with the following:

Step 3′: For each $j_1 \in X$ in the ascending order of $\Delta\tilde{z}^{\downarrow}_{j_1}(\boldsymbol{x})$, if $I'_2(\boldsymbol{x}) = \{j_2 \in (N \setminus X) \cap L'[j_1] \mid \Delta\tilde{z}_{j_1,j_2}(\boldsymbol{x}) < 0\} \neq \emptyset$ holds, choose $j_2 \in I'_2(\boldsymbol{x})$ with the minimum $\Delta\tilde{z}_{j_1,j_2}(\boldsymbol{x})$ and set $x_{j_1} \leftarrow 0$ and $x_{j_2} \leftarrow 1$. If the current solution $\boldsymbol{x}$ has been updated at least once in Step 3′, return to Step 1.

Step 4′: For each $j_1 \in X$ in the ascending order of $\Delta\tilde{z}_{j_1,\pi(j_1)}(\boldsymbol{x})$, if $I_4(\boldsymbol{x}) = \{j_3 \in X \cap L'[\pi(j_1)] \mid j_3 \neq j_1, \pi(j_3) \neq \pi(j_1), \Delta\tilde{z}_{j_1,\pi(j_1),j_3,\pi(j_3)}(\boldsymbol{x}) < 0\} \neq \emptyset$ holds, choose $j_3 \in I_4(\boldsymbol{x})$ with the minimum $\tilde{z}_{j_1,\pi(j_1),j_3,\pi(j_3)}(\boldsymbol{x})$ and set $x_{j_1} \leftarrow 0$, $x_{\pi(j_1)} \leftarrow 1$, $x_{j_3} \leftarrow 0$ and $x_{\pi(j_3)} \leftarrow 1$. If the current solution $\boldsymbol{x}$ has been updated at least once in Step 4′, return to Step 1; otherwise output $\boldsymbol{x}$ and exit.

Although a similar approach has been developed in local search algorithms for the Euclidean traveling salesman problem (TSP) that stores sorted lists containing only the k-nearest neighbors for each city in a geometric data structure called the k-dimensional tree (Johnson and McGeoch 1997), it is not suitable for finding the k-nearest neighbors efficiently in high-dimensional spaces. We thus extend it to be applicable to the high-dimensional column vectors $\boldsymbol{a}_j \in \{0,1\}^m$ ($j \in N$) of SPP by using a lazy construction algorithm for the neighbor list.

5 Adaptive Control of Penalty Weights

Recall that in our algorithm, solutions are evaluated by the alternative evaluation function $\tilde{z}(\boldsymbol{x})$ in which the fixed penalty weight vectors $\boldsymbol{w}^+, \boldsymbol{w}^-$ in the original objective function $z(\boldsymbol{x})$ are replaced with $\widetilde{\boldsymbol{w}}^+, \widetilde{\boldsymbol{w}}^-$, and the values of $\widetilde{w}_i^+$, $\widetilde{w}_i^-$ ($i \in M$) are adaptively controlled in the search. It is often reported that a single application of LS tends to stop at a locally optimal solution of insufficient quality when the large penalty weights are used. This is because it is often unavoidable to temporarily increase the values of some violations y_i^+ and y_i^- in order to reach an even better solution from a good solution through a sequence of neighborhood operations, and large penalty weights thus prevent LS from moving between such solutions. To overcome this, we incorporate an adaptive adjustment mechanism for determining appropriate values of penalty weights $\widetilde{w}_i^+$, $\widetilde{w}_i^-$ ($i \in M$) (Nonobe and Ibaraki 2001; Yagiura, Kishida, and Ibaraki 2006; Umetani, Arakawa, and Yagiura 2013). I.e., LS is applied iteratively while updating the values of the penalty weights $\widetilde{w}_i^+$, $\widetilde{w}_i^-$ ($i \in M$) after each call to LS. We call this sequence of calls to LS the weighting local search (WLS) according to (Selman and Kautz 1993; Thornton 2005). This strategy is also referred as the breakout algorithm (Morris 1993) and dynamic local search (Hutter, Tompkins, and Hoos 2002) in the literature.

Let $\boldsymbol{x}$ denote the solution at which the previous local search stops. We assume the original penalty weights w_i^+, w_i^- ($i \in M$) are sufficiently large. WLS resumes LS from $\boldsymbol{x}$ after updating the penalty weight vectors $\widetilde{\boldsymbol{w}}^+$, $\widetilde{\boldsymbol{w}}^-$. Starting from the original penalty weight vectors $(\widetilde{\boldsymbol{w}}^+, \widetilde{\boldsymbol{w}}^-) \leftarrow (\boldsymbol{w}^+, \boldsymbol{w}^-)$, the penalty weight vectors $\widetilde{\boldsymbol{w}}^+$, $\widetilde{\boldsymbol{w}}^-$ are updated as follows. Let $\boldsymbol{x}^*$ denote the best solution with respect to the original objective function $z(\boldsymbol{x})$ obtained so far. If $\tilde{z}(\boldsymbol{x}) \geq z(\boldsymbol{x}^*)$ holds, WLS uniformly decreases the penalty weights by $(\widetilde{\boldsymbol{w}}^+, \widetilde{\boldsymbol{w}}^-) \leftarrow \beta(\widetilde{\boldsymbol{w}}^+, \widetilde{\boldsymbol{w}}^-)$, where $0 < \beta < 1$ is a program parameter that is adaptively computed so that the new value of $\Delta\tilde{z}^{\downarrow}_j(\boldsymbol{x})$ becomes negative for 10% of variables satisfying $x_j = 1$. Otherwise, WLS increases the penalty weights by

$$\begin{aligned} \widetilde{w}_i^+ &\leftarrow \widetilde{w}_i^+ + \frac{z(\boldsymbol{x}^*) - \tilde{z}(\boldsymbol{x})}{\sum_{l \in M}(y_l^{+2} + y_l^{-2})}\, y_i^+, \quad i \in M, \\ \widetilde{w}_i^- &\leftarrow \widetilde{w}_i^- + \frac{z(\boldsymbol{x}^*) - \tilde{z}(\boldsymbol{x})}{\sum_{l \in M}(y_l^{+2} + y_l^{-2})}\, y_i^-, \quad i \in M. \end{aligned} \tag{4}$$

WLS iteratively applies LS, updating the penalty weight vectors $\widetilde{\boldsymbol{w}}^+$, $\widetilde{\boldsymbol{w}}^-$ after each call to LS until the time limit is

reached. Note that we modify 4-FNLS to evaluate solutions with both $\tilde{z}(\boldsymbol{x})$ and $z(\boldsymbol{x})$, and update the best solution $\boldsymbol{x}^*$ with respect to the original objective function $z(\boldsymbol{x})$ whenever an improved solution is found.

Algorithm WLS($\boldsymbol{x}$)

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

Output: The best solution $\boldsymbol{x}^*$ with respect to $z(\boldsymbol{x})$.

Step 1: Set $\boldsymbol{x}^* \leftarrow \boldsymbol{x}$, $\tilde{\boldsymbol{x}} \leftarrow \boldsymbol{x}$ and $(\widetilde{\boldsymbol{w}}^+, \widetilde{\boldsymbol{w}}^-) \leftarrow (\boldsymbol{w}^+, \boldsymbol{w}^-)$.

Step 2: Apply 4-FNLS$(\tilde{\boldsymbol{x}}, \widetilde{\boldsymbol{w}}^+, \widetilde{\boldsymbol{w}}^-)$ to obtain an improved solution $\tilde{\boldsymbol{x}}'$. Let $\boldsymbol{x}'$ be the best solution with respect to the original objective function $z(\boldsymbol{x})$ obtained during the call to 4-FNLS$(\tilde{\boldsymbol{x}}, \widetilde{\boldsymbol{w}}^+, \widetilde{\boldsymbol{w}}^-)$. Set $\tilde{\boldsymbol{x}} \leftarrow \tilde{\boldsymbol{x}}'$.

Step 3: If $z(\boldsymbol{x}') < z(\boldsymbol{x}^*)$ holds, then set $\boldsymbol{x}^* \leftarrow \boldsymbol{x}'$. If the time limit is reached, output $\boldsymbol{x}^*$ and halt.

Step 4: If $\tilde{z}(\tilde{\boldsymbol{x}}) \geq z(\boldsymbol{x}^*)$ holds, then uniformly decrease the penalty weights by $(\widetilde{\boldsymbol{w}}^+, \widetilde{\boldsymbol{w}}^-) \leftarrow \beta(\widetilde{\boldsymbol{w}}^+, \widetilde{\boldsymbol{w}}^-)$; otherwise, increase the penalty weight vectors $(\widetilde{\boldsymbol{w}}^+, \widetilde{\boldsymbol{w}}^-)$ by (4). Return to Step 2.

6 Computational Results

We report computational results for the 42 SPP instances from (Borndörfer 1998; Chu and Beasley 1998; Koch et al. 2011). Table 1 summarizes the information about the original and presolved instances. The second and fourth columns "#cst." shows the number of constraints, and the third and fifth columns "#var." shows the number of variables. Since several preprocessing techniques that often reduce the size of instances by removing redundant rows and columns are known (Borndörfer 1998), all algorithms were tested on the presolved instances. The instances marked with stars "⋆" are hard instances that cannot be solved optimally within at least 1h by the latest MIP solvers.

	Original		Presolved		Time
Instance	#cst.	#var.	#cst.	#var.	limit
aa01–06	675.3	7587.3	478.7	6092.7	600s
us01–04	121.3	295085.0	65.5	85772.5	600s
t0415–0421	1479.3	7304.3	820.7	2617.4	600s
⋆t1716–1722	475.7	58981.3	475.7	13193.6	3600s
v0415–0421	1479.3	30341.6	263.9	7277.0	600s
v1616–1622	1375.7	83986.7	1171.9	51136.7	600s
⋆ds	656	67732	656	67730	3600s
⋆ds-big	1042	174997	1042	173026	3600s
⋆ivu06-big	1177	2277736	1177	2197774	3600s
⋆ivu59	3436	2569996	3413	2565083	3600s

Table 1: Benchmark instances for SPP

We first compare our algorithm with two latest MIP solvers called CPLEX12.6 and SCIP3.1 (Achterberg 2009), and a local search solver called LocalSolver3.1 (Benoist et al. 2011). LocalSolver3.1 is not the latest version, but it performs better than the latest version 4.5 for SPP instances. These algorithms were tested on a MacBook Pro laptop computer with a 2.7 GHz Intel Core i7 processor, and were run on a single thread with time limits of 3600s for hard instances and 600s for other instances. We also compare our algorithm with a Lagrangian heuristic algorithm called the Wedelin heuristic (Wedelin 1995; Bastert, Hummel, and de Vries 2010). The computational results for the Wedelin heuristic are taken from (Bastert, Hummel, and de Vries 2010), where it was tested on a 1.3 GHz Sun UltraSPARC-IIIi processor and was run with a time limit of 600s.

Table 2 shows the relative gap (%) $\frac{z(\boldsymbol{x})-z_{\text{best}}}{z(\boldsymbol{x})} \times 100$ of the obtained feasible solutions under the original (hard) partitioning constraints, where z_{best} is the best upper bound among all algorithms and settings in this paper. The second column "z_{LP}" shows the optimal values of LP relaxation for SPP. The best upper bounds among the compared algorithms (or settings) are highlighted in bold. The numbers in parentheses show the number of instances for which the algorithm obtained at least one feasible solution, e.g., "(6/7)" shows that the algorithm obtained a feasible solution for six instances out of seven. Table 2 also summarizes the average performance of compared algorithms for both hard instances and other instances.

We observe that our algorithm achieves best upper bounds in 18 instances out of 42 for all instances, especially 7 instances out of 11 for hard instances. We note that local search algorithms and MIP solvers are quite different in character. Local search algorithms do not guarantee optimality because they typically search only a portion of the solution space. On the other hand, MIP solvers examine every possible solution, at least implicitly, in order to guarantee optimality. Hence, it is inherently difficult to find optimal solutions by local search algorithms even in the case of small instances, while MIP solvers find optimal solutions quickly for small instances and/or those having small gap between lower and upper bounds of optimal values. In view of these, our algorithm achieves sufficiently good upper bounds compared to the other algorithms for the benchmark instances, especially for the hard instances. We also note that there may still be room for large improvements in the case of hard instances because an optimal value of 93.52 was found for the instance "ds" by using a parallel extension of the latest MIP solvers called ParaSCIP on a supercomputer through a huge amount of computational effort (Shinano et al. 2013).

Table 3 shows the completion rate of the neighbor list in rows, i.e., the proportion of generated rows to all rows in the neighbor list. We observe that our algorithm achieves good performance while generating only a small part of the neighbor list for the large-scale "ds-big", "ivu06-big" and "ivu59" instances.

Table 4 shows the computational results of our algorithm for different settings. The left side of Table 4 shows the computational results of our algorithm for different neighbor list sizes, that is, when we stored the variables x_{j_2} with the top 1%, 10%, or 100% largest $|S_{j_1} \cap S_{j_2}|$ in the j_1th row of the neighbor list, respectively. The column "w/o" shows the results of our algorithm without the neighbor list. We observe that the performance of the local search algorithm is much improved by the neighbor list, and our algorithm attains good performance even if the size of the neighbor list is considerably small. The right side of Table 4 shows the

Instance	z_{LP}	z_{best}	CPLEX12.6	SCIP3.1	Wedelin	LocalSolver3.1	Proposed
aa01–06	40372.8	40588.8	**0.00**% (6/6)	**0.00**% (6/6)	—	13.89% (1/6)	1.95% (6/6)
us01–04	9749.4	9798.3	**0.00**% (4/4)	0.00% (3/4)	—	11.26% (2/4)	0.78% (4/4)
t0415–0421	5199083.7	5471010.9	**0.25**% (7/7)	1.13% (6/7)	1.30% (5/7)	∞ (0/7)	0.87% (6/7)
⋆t1716–1722	121445.8	162039.9	5.88% (7/7)	4.57% (7/7)	10.27% (7/7)	37.36% (1/7)	**1.25**% (7/7)
v0415–0421	2385764.2	2393130.0	**0.01**% (7/7)	**0.01**% (7/7)	0.71% (6/7)	0.06% (7/7)	0.02% (7/7)
v1616–1622	1021288.8	1025552.4	**0.00**% (7/7)	0.00% (7/7)	2.42% (3/7)	4.60% (7/7)	0.16% (7/7)
⋆ds	57.23	200.36	2.60%	36.44%	24.16%	84.15%	**0.00**%
⋆ds-big	86.82	876.67	55.13%	66.46%	—	91.24%	**0.00**%
⋆ivu06-big	135.43	168.17	19.83%	16.83%	—	51.93%	**0.00**%
⋆ivu59	884.46	1299.40	50.55%	57.01%	—	64.69%	**4.15**%
Avg. gap (w/o stars)			**0.06**% (31/31)	0.24% (29/31)	1.29% (14/21)	4.25% (17/31)	0.71% (30/31)
Avg. gap (with stars)			15.39% (11/11)	18.98% (11/11)	12.01% (8/ 8)	65.87% (5/11)	**1.17**% (11/11)

Table 2: Computational results of the latest solvers and the proposed algorithm for the benchmark instances

	Size of neighbor list				Type of neighborhood	
Instance	$\alpha = 0.01$	$\alpha = 0.1$	$\alpha = 1.0$	w/o	2-FNLS	4-FNLS
aa01–06	1.95% (6/6)	1.95% (6/6)	**1.79**% (6/6)	3.57% (6/6)	3.05% (6/6)	**1.95**% (6/6)
us01–04	0.83% (4/4)	**0.78**% (4/4)	1.14% (4/4)	1.01% (4/4)	**0.47**% (4/4)	0.78% (4/4)
t0415–0421	0.87% (6/7)	0.87% (6/7)	0.87% (6/7)	2.36% (2/7)	2.24% (6/7)	0.87% (6/7)
⋆t1716–1722	4.76% (7/7)	**1.25**% (7/7)	2.50% (7/7)	5.85% (7/7)	4.76% (7/7)	**1.25**% (7/7)
v0415–0421	0.01% (7/7)	0.02% (7/7)	**0.01**% (7/7)	0.03% (7/7)	**0.02**% (7/7)	0.02% (7/7)
v1616-1622	0.16% (7/7)	0.16% (7/7)	**0.15**% (7/7)	0.71% (7/7)	0.19% (7/7)	**0.16**% (7/7)
⋆ds	11.68%	**0.00**%	6.27%	14.94%	18.23%	**0.00**%
⋆ds-big	35.88%	**0.00**%	24.99%	28.97%	27.62%	**0.00**%
⋆ivu06-big	3.29%	**0.00**%	1.90%	18.64%	3.99%	**0.00**%
⋆ivu59	9.63%	4.15%	**0.00**%	33.40%	4.88%	**4.15**%
Avg. gap (w/o stars)	**0.71**% (30/31)	**0.71**% (30/31)	0.72% (30/31)	0.92% (26/31)	1.17% (30/31)	**0.71**% (30/31)
Avg. gap (with starts)	8.53% (11/11)	**1.17**% (11/11)	4.61% (11/11)	12.45% (11/11)	8.01% (11/11)	**1.17**% (11/11)

Table 4: Computational results of the proposed algorithm for different settings

Instance	1min	10min	30min	1h
aa01–06	33.15%	51.77%	—	—
us01–04	23.10%	27.76%	—	—
t0415–0421	89.75%	97.11%	—	—
⋆t1716–1722	38.00%	86.55%	96.29%	98.56%
v0415–0421	36.78%	39.44%	—	—
v1616-1622	5.28%	8.81%	—	—
⋆ds	1.37%	10.15%	22.22%	34.69%
⋆ds-big	0.13%	1.11%	3.27%	5.70%
⋆ivu06-big	0.01%	0.02%	0.08%	0.18%
⋆ivu59	0.01%	0.01%	0.04%	0.41%

Table 3: Completion rate of the neighbor list in rows

computational results of 2-FNLS and 4-FNLS. We observe that the 4-flip neighborhood search substantially improves the performance of our algorithm, even though it explores a quite limited space in the search.

7 Conclusion

We presented a data mining approach for reducing the search space of local search algorithms for large-scale SPPs. In this approach, we construct a k-nearest neighbor graph by extracting variable associations from the instance to be solved in order to identify promising pairs of flipping variables in the 2-flip neighborhood. We also developed a 4-flip neighborhood local search algorithm that flips four variables alternately along 4-paths or 4-cycles in the k-nearest neighbor graph. Comparison of computation with the latest solvers shows that our algorithm performs efficiently for large-scale SPP instances.

We note that these data mining approaches could also be beneficial for efficiently solving other large-scale combinatorial problems, particularly for hard instances having large gaps between the lower and upper bounds of the optimal values.

References

Achterberg, T.; Koch, T.; and Martin, A. 2006. MIPLIB2003. *Operations Research Letters* 34:361–372.

Achterberg, T. 2009. SCIP: Solving constraint integer programs. *Mathematical Programming Computation* 1:1–41.

Adenso-Díaz, B., and Laguna, M. 2006. Fine-tuning of algorithms using fractional experimental designs and local search. *Operations Research* 54:99–114.

Atamtürk, A.; Nemhauser, G. L.; and Savelsbergh, M. W. P. 1995. A combined Lagrangian, linear programming, and implication heuristic for large-scale set partitioning problems. *Journal of Heuristics* 1:247–259.

Balaprakash, P.; Birattari, M.; and Stützle, T. 2007. Improvement strategies for the F-race algorithm: Sampling design and iterative refinement. In *Proceedings of International Workshop on Hybrid Metaheuristics (HM)*, 108–122.

Barahona, F., and Anbil, R. 2000. The volume algorithm: Producing primal solutions with a subgradient method. *Mathematical Programming* A87:385–399.

Bastert, O.; Hummel, B.; and de Vries, S. 2010. A generalized Wedelin heuristic for integer programming. *INFORMS Journal on Computing* 22:93–107.

Benoist, T.; Estellon, B.; Gardi, F.; Megel, R.; and Nouioua, K. 2011. LocalSolver 1.x: A black-box local-search solver for 0-1 programming. *4OR* 9:299–316.

Borndörfer, R. 1998. *Aspects of set packing, partitioning and covering*. Ph.D. Dissertation, Technischen Universität, Berlin.

Boschetti, M. A.; Mingozzi, A.; and Ricciardelli, S. 2008. A dual ascent procedure for the set partitioning problem. *Discrete Optimization* 5:735–747.

Caprara, A.; Fischetti, M.; and Toth, P. 1999. A heuristic method for the set covering problem. *Operations Research* 47:730–743.

Ceria, S.; Nobili, P.; and Sassano, A. 1998. A Lagrangian-based heuristic for large-scale set covering problems. *Mathematical Programming* 81:215–228.

Chu, P. C., and Beasley, J. E. 1998. Constraint handling in genetic algorithms: The set partitioning problem. *Journal of Heuristics* 11:323–357.

Crawford, B.; Soto, R.; Monfroy, E.; Palma, W.; Castro, C.; and Paredes, F. 2013. Parameter tuning of a choice-function based hyperheuristic using particle swarm optimization. *Expert Systems with Applications* 40:1690–1695.

Hamadi, F.; Monfroy, E.; and Subion, F., eds. 2011. *Autonomous Search*. Springer.

Hutter, F.; Hoos, H. H.; and Leyton-Brown, K. 2009. ParamILS: An automated algorithm configuration framework. *Journal of Artificial Intelligence Research* 36:267–306.

Hutter, F.; Tompkins, D. A.; and Hoos, H. H. 2002. Scaling and probabilistic smoothing: Efficient dynamic local search for SAT. In *Proceedings of International Conference on Principles and Practice of Constraint Programming (CP)*, 233–248.

Johnson, D. S., and McGeoch, L. A. 1997. The traveling salesman problem: A case study. In Aarts, E., and Lenstra, K., eds., *Local Search in Combinatorial Optimization*. Princeton University Press. 215–310.

Kadioglu, S.; Malitsky, Y.; Sellmann, M.; and Tierney, K. 2010. ISAC – Instance-specific algorithm configuration. In *Proceedings of European Conference on Artificial Intelligence (ECAI)*, 751–756.

Kadioglu, S.; Malitsky, Y.; and Sellmann, M. 2012. Non-model-based search guidance for set partitioning problems. In *Proceedings of AAAI Conference on Artificial Intelligence (AAAI)*, 493–498.

KhudaBukhsh, A. R.; Xu, L.; Hoos, H.; and Leyton-Brown, K. 2009. SATenstein: Automatically building local search SAT solvers from components. In *Proceedings of International Joint Conference on Artificial Intelligence (IJCAI)*, 517–524.

Koch, T.; Achterberg, T.; Andersen, E.; Bastert, O.; Berthold, T.; Bixby, R. E.; Danna, E.; Gamrath, G.; Gleixner, A. M.; Heinz, S.; Lodi, A.; Mittlmann, H.; Ralphs, T.; Salvagnin, D.; Steffy, D. E.; and Wolter, K. 2011. MIPLIB2010: Mixed integer programming library version 5. *Mathematical Programming Computation* 3:103–163.

Linderoth, J. T.; Lee, E. K.; and Savelsbergh, M. W. P. 2001. A parallel, linear programming-based heuristic for large-scale set partitioning problems. *INFORMS Journal on Computing* 13:191–209.

Michel, L., and Van Hentenryck, P. 2000. Localizer. *Constraints* 5:43–84.

Monfroy, E.; Castro, C.; Crawford, B.; Soto, R.; Paredes, F.; and Figueroa, C. 2013. A reactive and hybrid constraint solver. *Journal of Experimental & Theoretical Artificial Intelligence* 25:1–22.

Morris, P. 1993. The breakout method for escaping from local minima. In *Proceedings of National Conference on Artificial Intelligence (AAAI)*, 40–45.

Nonobe, K., and Ibaraki, T. 2001. An improved tabu search method for the weighted constraint satisfaction problem. *INFOR* 39:131–151.

Pesant, G., and Gendreau, M. 1999. A constraint programming framework for local search methods. *Journal of Heuristics* 5:255–279.

Selman, B., and Kautz, H. 1993. Domain-independent extensions to GSAT: Solving large structured satisfiability problems. In *Proceedings of International Conference on Artificial Intelligence (IJCAI)*, 290–295.

Shaw, P. 2002. Improved local search for CP toolkits. *Annals of Operations Research* 115:31–50.

Shinano, Y.; Achterberg, T.; Berthold, T.; Heinz, S.; Koch, T.; and Winkler, M. 2013. Solving hard MIPLIB2003 problems with ParaSCIP on supercomputers: An update. Technical Report ZIB-Report 13-66, Zuse Institute Berlin.

Thornton, J. 2005. Clause weighting local search for SAT. *Journal of Automated Reasoning* 35:97–142.

Umetani, S.; Arakawa, M.; and Yagiura, M. 2013. A heuristic algorithm for the set multicover problem with generalized upper bound constraints. In *Proceedings of Learning and Intelligent Optimization Conference (LION)*, 75–80.

Van Hentenryck, P., and Michel, L. 2005. *Constraint-Based Local Search*. The MIT Press.

Voudouris, C.; Dorne, R.; Lesaint, D.; and Liret, A. 2001. iOpt: A software toolkit for heuristic search methods. In *Proceedings of Principles and Practice of Constraint Programming (CP)*, 716–729.

Wedelin, D. 1995. An algorithm for large scale 0-1 integer programming with application to airline crew scheduling. *Annals of Operations Research* 57:283–301.

Yagiura, M., and Ibaraki, T. 1999. Analysis on the 2 and 3-flip neighborhoods for the MAX SAT. *Journal of Combinatorial Optimization* 3:95–114.

Yagiura, M.; Kishida, M.; and Ibaraki, T. 2006. A 3-flip neighborhood local search for the set covering problem. *European Journal of Operational Research* 172:472–499.