A General Purpose Local Search Algorithm for Binary Optimization

Dimitris Bertsimas, Dan Iancu, Dmitriy Katz
Operations Research Center, Massachusetts Institute of Technology, E40-147, Cambridge, MA 02139, USA
{dbertsim@mit.edu, daniancu@mit.edu, dimdim@mit.edu}

We develop a general purpose local search algorithm for binary optimization problems, whose complexity and performance are explicitly controlled by a parameter Q, measuring the depth of the local search neighborhood. We show that the algorithm is pseudo-polynomial for general cost vector $\boldsymbol{c}$, and achieves a $\frac{w^2}{2w-1}$ approximation guarantee for set packing problems with exactly w ones in each column of the constraint matrix $\boldsymbol{A}$, when using $Q = w^2$. Most importantly, we find that the method has practical promise, as it delivers performance that is either comparable to or strictly better than leading optimization software (CPLEX 9.1) on large, randomly generated instances of both set covering and set packing problems.

Key words: Programming, Integer, Algorithms, Heuristic

---

1. Introduction

In the last fifty years there has been considerable progress in our ability to solve large scale binary optimization problems:

$$\begin{aligned} \max \quad & \boldsymbol{c}'\boldsymbol{x} \\ \text{s.t.} \quad & \boldsymbol{A}\boldsymbol{x} \leq \boldsymbol{b} \\ & \boldsymbol{x} \in \{0,1\}^n \\ & \boldsymbol{A} \in \mathbb{Z}^{m \cdot n},\ \boldsymbol{b} \in \mathbb{Z}^m,\ \boldsymbol{c} \in \mathbb{Z}^n. \end{aligned} \tag{1}$$

A testimony of this progress is the fact that major codes (like CPLEX and EXPRESS) are now capable of solving such problems that a decade ago were out of reach. In addition to very significant speedups in computing power, the two major ingredients that led to progress on the algorithmic side were: a) the introduction of new cutting plane methods, using a plethora of valid inequalities that improve the bounds on the solution and the ability to prove optimality, and b) the use of heuristic algorithms. While it is difficult to make an exact assessment of the merits of each algorithmic development, we believe that new cutting plane methods had a more significant impact than the use of heuristic methods.

Despite the major progress of the field, we still cannot solve especially dense binary problems. In real-world applications, there is stringent desire to find feasible solutions that improve current practice, without necessarily having a proof of their optimality. Thus, there is a definite need to develop general purpose methods producing high quality feasible solutions. There are relatively few methods for general binary integer programming problems (see Aarts and Lenstra (1997) for a review), including the lift-and-project methods (Balas et al., 1993), the pivot-and-complement heuristic (Balas and Martin, 1980), the "feasibility pump" (Fischetti et al., 2005), and the "pivot, cut and dive" heuristic (Eckstein and Nediak, 2007).

In this paper, we develop a new algorithm for general binary optimization problems, and provide both theoretical and empirical evidence for its strength. Specifically, our contributions are as follows:

1. The algorithm is genuinely general purpose, in that it does not utilize any special combinatorial structure in Problem (1).

2. The tradeoff between complexity and performance of the algorithm is explicitly controlled by a parameter Q, which intuitively measures the depth of the neighborhood in the local search. More precisely, with increasing Q, the algorithm can deliver higher quality solutions, at the expense of higher running time.

3. We show that the running time is bounded by $O\left(\|\boldsymbol{c}\|_1^2 \cdot n \cdot \binom{2m}{Q} \cdot \max(m,n)\right)$, i.e., for a fixed Q, the algorithm is pseudo-polynomial for general $\boldsymbol{c}$ and strongly polynomial when $\boldsymbol{c} = \boldsymbol{e}$, a vector of ones.

4. For the unweighted maximum w-set packing problem ($\boldsymbol{A} \in \{0,1\}^{m \cdot n}$, with w ones on each column, and $\boldsymbol{b} = \boldsymbol{c} = \boldsymbol{e}$), we show that our algorithm achieves a $\frac{w^2}{2w-1}$ approximation guarantee, contrasted with the slightly stronger bound of $\frac{w}{2}$ due to Hurkens and Schrijver (1989). Note that the latter heuristic is not general-purpose, but rather a specific result of a theorem pertaining to set packing problems.

5. Most importantly, we compare the algorithm's performance with CPLEX 9.1 on randomly generated instances for both set covering and set packing, with very encouraging results. Specifically, the proposed algorithm outperforms CPLEX after approximately 20 hours (when both methods are run with the same memory, 6GB), and sometimes even earlier.

The structure of rest of the paper is as follows. In Section 2, we present the algorithm, and give an example (which is further expanded in the Appendix). In Section 3, we analyze its running time, while in Section 4 we provide the theoretical guarantee for a class of set packing problems. In Section 5, we discuss implementation details, and in Section 6, we provide empirical evidence of the algorithm's strength by comparing its performance with CPLEX, for several classes of set covering and packing problems.

2. Algorithm

Our algorithm takes as inputs the matrix $\boldsymbol{A}$, the vectors $\boldsymbol{b}$ and $\boldsymbol{c}$, a parameter Q and an initial feasible solution $\boldsymbol{z}_0$, and constructs a sequence of feasible solutions $\boldsymbol{z}$ with monotonically increasing objective function values. The parameter Q controls the tradeoff between the quality of the final output solution and the computational complexity of the algorithm.

2.1. Notation

For any vector $\boldsymbol{x} \in \{0,1\}^n$, we define the following:

- $\boldsymbol{x}_v = \max(\boldsymbol{A}\boldsymbol{x} - \boldsymbol{b}, \boldsymbol{0}) \in \mathbb{Z}_+^m$: the amount of constraint violation produced by $\boldsymbol{x}$.
- $\boldsymbol{x}_u = \max(\boldsymbol{b} - \boldsymbol{A}\boldsymbol{x}, \boldsymbol{0}) \in \mathbb{Z}_+^m$: the amount of constraint "looseness" created by $\boldsymbol{x}$.
- $\boldsymbol{x}_w = \min(\boldsymbol{x}_u, \boldsymbol{e}) \in \{0,1\}^m$.
- $trace(\boldsymbol{x}) = [\boldsymbol{x}_v; \boldsymbol{x}_w] \in \mathbb{Z}_+^m \times \{0,1\}^m$.

Furthermore, we introduce the following concepts:

- Two solutions $\boldsymbol{x}$ and $\boldsymbol{y}$ are said to be *adjacent* if $\boldsymbol{e}'|\boldsymbol{x} - \boldsymbol{y}| = 1$.
- A feasible solution $\boldsymbol{z}_1$ is said to be *better* than another feasible solution $\boldsymbol{z}_2$ if $\boldsymbol{c}'\boldsymbol{z}_1 > \boldsymbol{c}'\boldsymbol{z}_2$.
- Let $\boldsymbol{z}$ be the best feasible solution available to the algorithm at a particular iteration. A solution $\boldsymbol{y}$ is called *interesting* if the following three properties hold:

(A1) $\|\boldsymbol{y}_v\|_\infty \leq 1$: no constraint is violated by more than one unit.

(A2) $\|trace(\boldsymbol{y}) - trace(\boldsymbol{z})\|_1 \leq Q$: the total amount of violation in $\boldsymbol{y}$ plus the number of different loose constraints (as compared to $\boldsymbol{z}$) is at most Q.

(A3) $\boldsymbol{c}'\boldsymbol{y} > \boldsymbol{c}'\boldsymbol{x}\,, \forall\, \boldsymbol{x}$ already examined by the algorithm, satisfying $h\,(\mathit{trace}(\boldsymbol{x})) = h\,(\mathit{trace}(\boldsymbol{y}))$.

Here, $h : \{0,1\}^{2m} \rightarrow \mathbb{N}$ is a function mapping traces of interesting solutions[1] into integers. The only restriction we impose on $h(\cdot)$ is that evaluating it should be linear in the size of the input: $O(m)$. Apart from that, it can be injective, in which case only solutions with identical traces will be compared, or it can be a hash function (for an introduction to hash functions, see Cormen et al. (2001)). The reason for introducing such a hash function is to accelerate the algorithm, at the potential expense of worsening the performance. We will elaborate more on this tradeoff in Section 5, which is dedicated to implementation details.

Note that due to Condition (A3), for every value i in the range of h, the algorithm needs to store the highest objective function value of an interesting solution $\boldsymbol{x}$ satisfying $h(\mathit{trace}(\boldsymbol{x})) = i$. We will refer to the location where this value is stored as the *trace box* ($\mathcal{TB}$) corresponding to $\boldsymbol{x}$ or to $\mathit{trace}(\boldsymbol{x})$, and will denote it by $\mathcal{TB}[i]$.

- The set of interesting solutions is also referred to as the *solution list* ($\mathcal{SL}$).

2.2. Algorithm Outline

With these definitions, we now give a brief outline of the algorithm, which will also give some insight into the types of data structures that are needed. The key ingredient in the heuristic are interesting solutions. In a typical iteration, the algorithm will pick a candidate $\boldsymbol{x}$ from the list of interesting solutions ($\mathcal{SL}$), and examine all solutions $\boldsymbol{y}$ adjacent to it. If these solutions turn out to be interesting themselves, they are stored in the list, and the appropriate trace boxes are changed.

By following this method, occasionally we come across solutions $\boldsymbol{y}$ which are feasible. If they are also better than the best current feasible solution $\boldsymbol{z}$, then $\boldsymbol{z}$ is replaced, the list and the trace boxes are cleared, and the procedure resumes by examining solutions adjacent to $\boldsymbol{z}$. A formal statement follows.

To understand the steps in the algorithm, let us consider the following example of a set

[1] Since $h(\cdot)$ is only applied to *interesting* solutions $\boldsymbol{y}$, which, by condition (A1), must satisfy $\boldsymbol{y}_v \in \{0,1\}^m$, we can take $h : \{0,1\}^{2m}$, instead of $h : \mathbb{Z}_+^m \times \{0,1\}^m$.

Algorithm 1: Local search heuristic
Input: matrix $\boldsymbol{A}$; vectors $\boldsymbol{b}, \boldsymbol{c}$; feasible solution $\boldsymbol{z}_0$; scalar parameter $Q > 0$
Output: Feasible solution $\boldsymbol{z}$ such that $\boldsymbol{c}'\boldsymbol{z} \geq \boldsymbol{c}'\boldsymbol{z}_0$
OPTIMIZEIP$(\boldsymbol{A}, \boldsymbol{b}, \boldsymbol{c}, \boldsymbol{z}_0, Q)$
(1) $\quad \boldsymbol{z} := \boldsymbol{z_0}; \quad \mathcal{SL} := \{\boldsymbol{z}\}$
(2) $\quad$ **while** $(\mathcal{SL} \neq \emptyset)$
(3) $\quad\quad$ get a new solution $\boldsymbol{x}$ from $\mathcal{SL}$
(4) $\quad\quad$ **foreach** ($\boldsymbol{y}$ adjacent to $\boldsymbol{x}$)
(5) $\quad\quad\quad$ **if** ($\boldsymbol{y}$ is feasible) **and** $(\boldsymbol{c}'\boldsymbol{y} > \boldsymbol{c}'\boldsymbol{z})$
(6) $\quad\quad\quad\quad \boldsymbol{z} \leftarrow \boldsymbol{y}$
(7) $\quad\quad\quad\quad \mathcal{SL} \leftarrow \emptyset; \quad \mathcal{TB}[\,i\,] \leftarrow -\infty, \ \forall i$
(8) $\quad\quad\quad\quad \mathcal{SL} \leftarrow \mathcal{SL} \cup \{\boldsymbol{y}\}$
(9) $\quad\quad\quad\quad$ **goto** Step 3
(10) $\quad\quad\quad$ **else if** ($\boldsymbol{y}$ is interesting)
(11) $\quad\quad\quad\quad \mathcal{TB}\,[\,h(\mathit{trace}(\boldsymbol{y}))\,] \leftarrow \boldsymbol{c}'\boldsymbol{y}$
(12) $\quad\quad\quad\quad \mathcal{SL} \leftarrow \mathcal{SL} \cup \{\boldsymbol{y}\}$
(13) $\quad$ **return** $\boldsymbol{z}$

packing problem:

$$\begin{array}{lrrrl} \max & x_1 & +x_2 & +x_3 & \\ \text{s.t.} & x_1 & & +x_3 & \leq 1 \\ & & x_2 & +x_3 & \leq 1 \\ & & x_1, x_2, x_3 \in \{0,1\}. & & \end{array} \tag{2}$$

It is easy to see, by inspection, that the optimal solution is $\boldsymbol{x}_{\text{opt}} \stackrel{\text{def}}{=} [\,x_1, x_2, x_3\,] = [\,1, 1, 0\,]$. In order to illustrate the steps that Algorithm 1 would take in finding this solution, we will make the following choice concerning the parameters and implementation:

- We will make the simplest possible run, with a parameter $Q = 1$.
- We will start the algorithm with the initial solution $\boldsymbol{z}_0 = [\,0, 0, 0\,]$.
- Since every trace of an interesting solution $\boldsymbol{x}$ is a binary vector, $\mathit{trace}(\boldsymbol{x}) \equiv [\,t_{2m-1}, t_{2m-2}, \ldots, t_1, t_0\,] \in \{0,1\}^{2m}$, we will take the mapping $h(\cdot)$ to be the decimal representation:

$$h : \{0,1\}^{2m} \rightarrow \mathbb{R}, \ h\left([\,t_{2m-1}, \ldots, t_1, t_0\,]\right) = \sum_{i=0}^{2m-1} t_i \cdot 2^i$$

- We will assume that the solution list $\mathcal{SL}$ is implemented as a First-In, First-Out (FIFO) list, so that solutions are extracted in the same order in which they are inserted.

With these remarks, we now proceed to list the first few steps in the algorithm:

- (Step 1) $\boldsymbol{z} := [0, 0, 0]$; $\mathcal{SL} := \{\,[0, 0, 0]\,\}$.

- (Step 2) $\mathcal{SL} := \{\, [0,0,0] \,\} \neq \emptyset$.
- (Step 3) $\boldsymbol{x} \leftarrow [0,0,0]$. Adjacent solutions are $[1,0,0], [0,1,0], [0,0,1]$.
  - (Step 4) $\boldsymbol{y} = [1,0,0]$, $trace(\boldsymbol{y}) = [0,0\,;\,0,1]$.
    - (Step 5) $\boldsymbol{y}$ feasible, $\boldsymbol{e}'\boldsymbol{y} = 1 > \boldsymbol{e}'\boldsymbol{z}$.
    - (Steps 6 - 8) $\boldsymbol{z} \leftarrow [1,0,0]$; $\mathcal{SL} \leftarrow \{\, [1,0,0] \,\}$; $\mathcal{TB}[\,i\,] \leftarrow -\infty$, $\forall i$.
- (Step 3) $\boldsymbol{x} \leftarrow [1,0,0]$. Adjacent solutions are $[0,0,0], [1,1,0], [1,0,1]$.
  - (Step 4) $\boldsymbol{y} = [0,0,0]$, $trace(\boldsymbol{y}) \stackrel{\text{def}}{=} [\boldsymbol{y}_v\,;\,\boldsymbol{y}_w] = [0,0;1,1]$.
    - (Step 5) $\boldsymbol{y}$ feasible, but $\boldsymbol{e}'\boldsymbol{y} = 1 = \boldsymbol{e}'\boldsymbol{z}$.
    - (Step 10) $\boldsymbol{y}$ is found to be interesting, since:
      (A1) true: $\|\boldsymbol{y}_v\|_\infty = \|[0,0]\|_\infty \leq 1$.
      (A2) true: $\|trace(\boldsymbol{y}) - trace(\boldsymbol{z})\|_1 \leq Q$.
      (A3) true: $\boldsymbol{e}'\boldsymbol{y} = 1 > \mathcal{TB}\,[h(trace(\boldsymbol{y}))] = \mathcal{TB}[3] = -\infty$.
    - (Steps 11 - 12) $\mathcal{TB}[3] \leftarrow 1$; $\mathcal{SL} \leftarrow \{\, [0,0,0] \,\}$.
  - (Step 4) $\boldsymbol{y} = [1,1,0]$, $trace(\boldsymbol{y}) = [0,0;0,0]$.
    - (Step 5) $\boldsymbol{y}$ feasible, $\boldsymbol{e}'\boldsymbol{y} = 2 > \boldsymbol{e}'\boldsymbol{z}\,(= 1)$.
    - (Steps 6 - 8) $\boldsymbol{z} \leftarrow [1,1,0]$; $\mathcal{SL} \leftarrow \{\, [1,1,0] \,\}$; $\mathcal{TB}[\,i\,] \leftarrow -\infty$, $\forall i$.

We note that, although the algorithm has found the optimal solution $\boldsymbol{z} = \boldsymbol{x}_{\text{opt}} = [1,1,0]$, quite a few steps remain, which we have listed, for completeness, in the Appendix. Moreover, the particular choices of implementation in the above example have been made in order to facilitate exposition, and are by no means efficient. In Section 5, we include a detailed discussion of the data structures and hash functions used in our implementation.

3. Running Time

In this section, we bound the running time of the algorithm as follows:

Theorem 1. *For fixed Q and injective $h(\cdot)$, the running time of Algorithm 1 is bounded above by:*

$$O\left(\|\boldsymbol{c}\|_1^2 \cdot n \cdot \binom{2m}{Q} \cdot \max(m,n)\right). \tag{3}$$

We postpone the proof of Theorem 1 until the end of this section, and first introduce the following lemma:

Lemma 1. *The total number of solutions $\boldsymbol{x}$ that can be examined between two successive updates of the current feasible solution $\boldsymbol{z}$ is $O\left(\binom{2m}{Q} \cdot \|\boldsymbol{c}\|_1\right)$.*

Proof. First note that whenever the current feasible solution $\boldsymbol{z}$ is updated, the solution list $\mathcal{SL}$ is emptied, the trace boxes are cleared, and only $\boldsymbol{z}$ is inserted in $\mathcal{SL}$. Hence for any solution $\boldsymbol{x} \neq \boldsymbol{z}$ to be examined, it must first be inserted into $\mathcal{SL}$.

By Condition (A3) in the definition of interesting solutions, an *interesting* $\boldsymbol{x}$ inserted into $\mathcal{SL}$ must satisfy $\boldsymbol{c}'\boldsymbol{x} > \mathcal{TB}[\,i\,]$, where $i = h\,(\mathit{trace}(\boldsymbol{x}))$. Since $\boldsymbol{x} \in \{0,1\}^n$, $\boldsymbol{c}'\boldsymbol{x} \in \left\{\sum_{c_i<0} c_i, \ldots, \sum_{c_i>0} c_i\right\}$. Hence the number of updates for any trace box i is at most $\|\boldsymbol{c}\|_1+1$, which implies that at most $\|\boldsymbol{c}\|_1 + 1$ different solutions $\boldsymbol{x}$ mapping to i can be inserted into $\mathcal{SL}$.

The number of trace boxes i is upper-bounded by the number of distinct traces of interesting solutions. If $\boldsymbol{x}$ is an interesting solution, then:

- Condition (A1) $\Rightarrow \|\boldsymbol{x}_v\|_\infty \leq 1 \Rightarrow \boldsymbol{x}_v \in \{0,1\}^m \Rightarrow \mathit{trace}(\boldsymbol{x}) \in \{0,1\}^{2m}$
- Condition (A2) $\Rightarrow \|\mathit{trace}(\boldsymbol{x}) - \mathit{trace}(\boldsymbol{z})\|_1 = \|\boldsymbol{x}_v - \boldsymbol{0}\|_1 + \|\boldsymbol{x}_w - \boldsymbol{z}_w\|_1 \leq Q$.

The number of binary vectors of length $2m$ satisfying this property is upper bounded by:

$$\binom{2m}{Q} + \binom{2m}{Q-1} + \cdots + \binom{2m}{1} + 1. \tag{4}$$

Thus, there are $O\left(\binom{2m}{Q}\right)$ trace boxes to keep track of. Since for each trace box at most $\|\boldsymbol{c}\|_1 + 1$ solutions can be inserted in $\mathcal{SL}$, we conclude that the number of solutions which can be examined, which is always less than the number of solutions inserted in the list, is $O\left(\binom{2m}{Q} \cdot \|\boldsymbol{c}\|_1\right)$. □

The following lemma deals with the amount of computation performed when examining an interesting solution $\boldsymbol{x}$.

Lemma 2. *The number of operations performed for any interesting solution $\boldsymbol{x}$ that is examined between two consecutive updates of the current feasible solution $\boldsymbol{z}$ is $O\,(n \cdot \max(m,n))$.*

Proof. Without going into the details of the implementation, let us consider what operations are performed when examining an interesting solution $\boldsymbol{x}$.

(B1) *trace*($\boldsymbol{x}$) is calculated. This implies the following:

- Computing $\boldsymbol{Ax} - \boldsymbol{b}$, which requires $O(m \cdot n)$ operations for a dense matrix $\boldsymbol{A}$.
- Comparing $\boldsymbol{Ax} - \boldsymbol{b}$ with $\boldsymbol{0}$, to check for violated or loose constraints, requiring $O(m)$ computations.

(B2) Computing the objective function for x, requiring $O(n)$ operations.

(B3) Examining all the solutions $\boldsymbol{y}$ adjacent to $\boldsymbol{x}$. One such examination entails:

- Computing *trace*($\boldsymbol{y}$) from *trace*($\boldsymbol{x}$). Since $\boldsymbol{y} = \boldsymbol{x} \pm \boldsymbol{e}_i \Rightarrow \boldsymbol{Ay} - \boldsymbol{b} = \boldsymbol{Ax} - \boldsymbol{b} \pm \boldsymbol{A_i}$. Because $\boldsymbol{Ax} - \boldsymbol{b}$ is already available, computing *trace*($\boldsymbol{y}$) only requires $O(m)$ operations.
- Computing the trace box for $\boldsymbol{y}$, $\mathcal{TB}[\, h(\textit{trace}(\boldsymbol{y}))\,]$. As mentioned earlier, we are requiring that an evaluation of the function $h(\cdot)$ should use $O(m)$ operations, i.e., linear in the size of the argument. Thus $\mathcal{TB}[\, h(\textit{trace}(\boldsymbol{y}))\,]$ can be computed with $O(m)$ operations.
- Computing the objective function value for $\boldsymbol{y}$. This is $O(1)$, since $\boldsymbol{c}'\boldsymbol{y} = \boldsymbol{c}'\boldsymbol{x} \pm \boldsymbol{c}_i$.
- Comparing $\boldsymbol{c}'\boldsymbol{y}$ with $\mathcal{TB}[\, h(\textit{trace}(\boldsymbol{y}))\,]$. Since the theorem assumes that the current feasible solution $\boldsymbol{z}$ is not updated, the results of the examination could be that (*i*) $\boldsymbol{y}$ is ignored or (*ii*) $\mathcal{TB}[\, h(\textit{trace}(\boldsymbol{y}))\,]$ is replaced and $\boldsymbol{y}$ is added to $\mathcal{SL}$. Overall complexity is at most $O(n)$.

Since the number of solutions $\boldsymbol{y}$ adjacent to a given $\boldsymbol{x}$ is n, the overall complexity of step (B3) above is $O\left(n \cdot \max(m, n)\right)$, and dominates steps (B1) and (B2). We conclude that the overall complexity associated with examining any interesting solution $\boldsymbol{x}$ is $O\left(n \cdot \max(m, n)\right)$. □

With the help of the preceding lemmas, we can now prove Theorem 1.

Proof. From Lemma 1, the number of solutions that have to be examined between two successive updates of $\boldsymbol{z}$ is $O\left(\binom{2m}{Q} \cdot \|\boldsymbol{c}\|_1\right)$. From Lemma 2, each such examination entails $O\left(n \cdot \max(m, n)\right)$ operations. Hence the amount of operations that are performed while examining interesting solutions between updates of $\boldsymbol{z}$ is $O\left(\binom{2m}{Q} \cdot \|\boldsymbol{c}\|_1 \cdot n \cdot \max(m, n)\right)$.

Each update of the current feasible solution $\boldsymbol{z}$ involves copying the new solution ($O(n)$), emptying the solution list and clearing the trace boxes. The latter operations are linear in

the total number of trace boxes, which, from a result in Lemma 1, is $O\left(\binom{2m}{Q}\right)$. Therefore updating $\boldsymbol{z}$ entails $O\left(\max\left\{n, \binom{2m}{Q}\right\}\right)$ operations.

Since $\boldsymbol{z} \in \{0,1\}^n \Rightarrow \boldsymbol{c}'\boldsymbol{z} \in \left\{\sum_{c_i<0} c_i, \ldots, \sum_{c_i>0} c_i\right\}$. Hence, there can be at most $\|\boldsymbol{c}\|_1 + 1$ updates of $\boldsymbol{z}$. Therefore, the total running time of the algorithm is:

$$O\left(\|\boldsymbol{c}\|_1 \left[\binom{2m}{Q} \cdot \|\boldsymbol{c}\|_1 \cdot n \cdot \max(m,n) + \max\left\{n, \binom{m}{Q}\right\}\right]\right) =$$
$$= O\left(\|\boldsymbol{c}\|_1^{\,2} \cdot n \cdot \binom{2m}{Q} \cdot \max(m,n)\right) \qquad \square$$

We make the observation that when $\boldsymbol{c} = \boldsymbol{e}$, the above result becomes $O\left(n^3 \cdot \binom{2m}{Q} \cdot \max(m,n)\right)$, proving that Algorithm 1 is strongly polynomial for a fixed Q.

4. Performance Guarantee for Set Packing Problems

So far, we have put no restrictions on the particular data-structures that are used. While this level of generality was appropriate for the algorithm description, in order to prove a meaningful result about the performance, we have to be more specific about the details.

As such, for the remaining part of this section, we consider a solution list $\mathcal{SL}$ implemented as a First In First Out (FIFO) list, and we consider the ideal case of an injective $h(\cdot)$, namely when each trace box corresponds to a unique trace of an interesting solution, and, implicitly, only solutions having exactly the same trace are compared.

We focus on the following binary optimization problem, which is an integer programming formulation for the well known unweighted w-set packing problem:

$$\begin{aligned} \max \quad & \boldsymbol{e}'\boldsymbol{x} \\ \text{s.t.} \quad & \boldsymbol{A}\boldsymbol{x} \leq \boldsymbol{e} \\ & \boldsymbol{x} \in \{0,1\}^n \\ & \boldsymbol{A} \in \{0,1\}^{m \cdot n}, \end{aligned} \tag{5}$$

with the additional restriction that each variable x_i should participate in exactly w constraints:

$$\boldsymbol{e}'\boldsymbol{A} = w\boldsymbol{e}'. \tag{6}$$

Let $\boldsymbol{z}^*$ denote an optimal solution to Problem (5) and $Z^* = \boldsymbol{e}'\boldsymbol{z}^*$ be its associated objective function value. Then the following theorem holds:

Theorem 2. *If $Q = w^2$ and $w > 1$, Algorithm 1, operating with a FIFO list and an injective $h(\cdot)$, finds a feasible solution $\boldsymbol{z}$ for Problem (5) with objective function value $Z_H = \boldsymbol{e}'\boldsymbol{z}$ satisfying:*

$$\frac{Z^*}{Z_H} \leq \frac{w^2}{2w-1}. \tag{7}$$

We defer the proof of the theorem to the end of the section, and first introduce a lemma summarizing the properties of a feasible $\boldsymbol{z}$ not satisfying the requirement of Theorem 2. In what follows, $\boldsymbol{a}_j^{'}$ will always denote the j-th row of the matrix $\boldsymbol{A}$, $\boldsymbol{e}_i$ will denote the i-th unit vector, and $\boldsymbol{e}$ will denote the vector with 1 in every component.

Lemma 3. *Let $\boldsymbol{z}$ be a feasible solution such that $\boldsymbol{e}'\boldsymbol{z} < \frac{2w-1}{w^2} Z^*$ and $\boldsymbol{e}'\boldsymbol{z} \geq \boldsymbol{e}'\boldsymbol{y}$, for all solutions $\boldsymbol{y}$ feasible and adjacent to $\boldsymbol{z}$. Also let*

$$\mathcal{O} = \{i \in \{1,\ldots,n\} \ : \ \boldsymbol{z}_i^* = 1\}, \quad \textit{(components} = 1 \textit{ in the optimal solution)} \tag{8}$$

$$\mathcal{I} = \{i \in \{1,\ldots,n\} \ : \ \boldsymbol{z}_i = 1\}, \quad \textit{(components} = 1 \textit{ in the current solution)} \tag{9}$$

$$\mathcal{V}_i = \left\{l \in \{1,\ldots,m\} \ : \ \boldsymbol{a}_l^{'}(\boldsymbol{z}+\boldsymbol{e}_i) > 1\right\}, \ \ (i \in \mathcal{O}), \tag{10}$$

(constraints violated by increasing i-th component of current solution)

$$\mathcal{V} = \{i \in \mathcal{O} \ : \ |\mathcal{V}_i| = 1\}, \tag{11}$$

$$R = |\mathcal{V}|; \quad \mathcal{V} \equiv \{v_1,\ldots,v_R\}. \tag{12}$$

Then the following properties hold:

$$\boldsymbol{A}(\boldsymbol{e}_i+\boldsymbol{e}_j) \leq \boldsymbol{e} \textit{ and } \mathcal{V}_i \cap \mathcal{V}_j = \emptyset, \quad \forall\, i \neq j \in \mathcal{O}, \tag{13}$$

$$R > \frac{Z^*}{w}, \tag{14}$$

$$v_i \in \mathcal{V} \quad \Rightarrow \quad v_i \notin \mathcal{I} \quad \textit{and} \quad \exists\, p_i \in \mathcal{I} \setminus \mathcal{O} \ \textit{ s.t. } \ \boldsymbol{A}(\boldsymbol{z}+\boldsymbol{e}_{v_i}-\boldsymbol{e}_{p_i}) \leq \boldsymbol{e}, \tag{15}$$

$$\exists\, j \in \{1,\ldots,m\} \textit{ and } \exists\, T \leq R \quad \textit{s.t.} \quad \boldsymbol{A}\left(\boldsymbol{z} + \sum_{i=1}^{T} \boldsymbol{e}_{v_i} - \sum_{i=1}^{T} \boldsymbol{e}_{p_i} + \boldsymbol{e}_j\right) \leq \boldsymbol{e}. \tag{16}$$

Proof. From the definition of $\mathcal{O}$, $\boldsymbol{z}^* = \sum_{i\in\mathcal{O}} \boldsymbol{e}_i$. Since $\boldsymbol{z}^*$ feasible, $\boldsymbol{A}\,\boldsymbol{z}^* \leq \boldsymbol{e}$. With $\boldsymbol{A} \in \{0,1\}^{m\cdot n} \Rightarrow \boldsymbol{A}(\boldsymbol{e}_i+\boldsymbol{e}_j) \leq \boldsymbol{e}$, $\forall\, i \neq j \in \mathcal{O}$. Intuitively, this means that two variables, x_i and x_j, cannot participate in the same constraint.

To prove the second part of (13), assume, for the purposes of a contradiction, that $\exists\, l \in \mathcal{V}_i \cap \mathcal{V}_j \Rightarrow \boldsymbol{a}_l^{'}(\boldsymbol{z}+\boldsymbol{e}_i) > 1$ and $\boldsymbol{a}_l^{'}(\boldsymbol{z}+\boldsymbol{e}_j) > 1$. Since $\boldsymbol{z}$ feasible, $\boldsymbol{a}_l^{'}\boldsymbol{z} \leq 1 \Rightarrow \boldsymbol{a}_l^{'}\boldsymbol{e}_i = \boldsymbol{a}_l^{'}\boldsymbol{e}_j = 1$, in contradiction with the result in the previous paragraph.

To prove (14), first note that only constraints $\boldsymbol{a}_l$ that are tight at $\boldsymbol{z}$ can belong to $\mathcal{V}_i$:

$$\forall\, i \in \mathcal{O}, \forall\, l \in \mathcal{V}_i,\ \ \boldsymbol{a}_l^{'}\,(\boldsymbol{z}+\boldsymbol{e}_i) > 1 \ \Rightarrow\ \Big(\text{since } \boldsymbol{a}_j^{'}\,\boldsymbol{e}_i \leq 1,\ \forall\, j\Big) \ \Rightarrow\ \boldsymbol{a}_l^{'}\,\boldsymbol{z} = \boldsymbol{a}_l^{'}\,\boldsymbol{e}_i = 1. \quad (17)$$

Since each variable participates in exactly w constraints, and $\boldsymbol{e}'\boldsymbol{z} < \frac{2w-1}{w^2}\,Z^*$, the number of constraints that are tight at $\boldsymbol{z}$ always satisfies:

$$(\#\text{ constraints tight at } \boldsymbol{z}) \ <\ w \cdot \frac{2w-1}{w^2}\,Z^* = \left(2 - \frac{1}{w}\right) Z^*\,. \quad (18)$$

Now consider the sets $\mathcal{V}_i$. Since $\boldsymbol{e}'\boldsymbol{z}^* = Z^*$, there are Z^* such sets, one for each $i \in \mathcal{O}$. If $\exists\, i \in \mathcal{O}$ s.t. $\mathcal{V}_i = \emptyset$, then $\boldsymbol{z}+\boldsymbol{e}_i$ would be feasible, with a strictly larger objective function than $\boldsymbol{z}$, in contradiction with the second assumption concerning $\boldsymbol{z}$. Therefore $|\mathcal{V}_i| \geq 1, \forall\, i \in \mathcal{O}$, implying:

$$\sum_{i=1}^{Z^*} |\mathcal{V}_i| = \sum_{i:|\mathcal{V}_i|=1} + \sum_{i:|\mathcal{V}_i|\geq 2} \geq R + 2(Z^* - R) = 2\,Z^* - R. \quad (19)$$

We have argued that only constraints that $\boldsymbol{z}$ satisfies with equality can belong to $\mathcal{V}_i$. Thus, from (18) and (19) we obtain the desired relation (14):

$$2\,Z^* - R \leq \sum_{i=1}^{Z^*} |\mathcal{V}_i| < Z^* \left(2 - \frac{1}{w}\right) \Leftrightarrow R > \frac{Z^*}{w}.$$

To prove (15), observe that if $v_i \in \mathcal{V}$, then $v_i \in \mathcal{O}$ and $|\mathcal{V}_{v_i}| = 1$. Then (17) implies that $\exists$ unique $l \in \mathcal{V}_{v_i}$ s.t. $\boldsymbol{a}_l'\,(\boldsymbol{z}+\boldsymbol{e}_{v_i}) > 1$ and $\forall\, j \neq l,\ \boldsymbol{a}_j'\,(\boldsymbol{z}+\boldsymbol{e}_{v_i}) \leq 1$.

Assume $v_i \in \mathcal{I}$. Then $\boldsymbol{z} \geq \boldsymbol{e}_{v_i}$. Since each variable participates in w constraints, $\exists\ l_1, \ldots, l_w$ distinct constraints s.t. $\boldsymbol{a}_{l_j}^{'}\,\boldsymbol{e}_{v_i} = 1$, which implies $\boldsymbol{a}_{l_j}^{'}\,(\boldsymbol{z}+\boldsymbol{e}_{v_i}) \geq 2,\ \forall\, j = 1, \ldots, w$, in contradiction with $|\mathcal{V}_{v_i}| = 1$. Therefore, $v_i \notin \mathcal{I}$.

Consider again the unique l s.t. $\boldsymbol{a}_l^{'}\,(\boldsymbol{z}+\boldsymbol{e}_{v_i}) > 1$. From (17), $\boldsymbol{a}_l^{'}\,\boldsymbol{z} = 1 \Rightarrow \exists\, p_i \in \mathcal{I}$ s.t. $\boldsymbol{a}_l^{'}\,\boldsymbol{e}_{p_i} = 1$. Also, since $v_i \notin \mathcal{I}$, $p_i \neq v_i$. Assume $p_i \in \mathcal{O}$; then $\boldsymbol{a}_l^{'}\,\boldsymbol{e}_{p_i} = \boldsymbol{a}_l^{'}\,\boldsymbol{e}_{v_i} = 1,\ p_i, v_i \in \mathcal{O}$, in direct contradiction with (13). Hence $p_i \notin \mathcal{O}$.

Now consider $\hat{\boldsymbol{z}} = \boldsymbol{z} + \boldsymbol{e}_{v_i} - \boldsymbol{e}_{p_i}$. $\forall\, j \in \{1, \ldots, m\},\ j \neq l,\ \boldsymbol{a}_j^{'}\,\hat{\boldsymbol{z}}\ \leq \boldsymbol{a}_j^{'}\,(\boldsymbol{z}+\boldsymbol{e}_{v_i}) \leq 1$. Also, $\boldsymbol{a}_l^{'}\,\hat{\boldsymbol{z}} = 1 + 1 - 1 = 1$. Therefore, $\hat{\boldsymbol{z}}$ is feasible, concluding the last part of (15).

Before establishing the proof of (16), first note that result (15) can be extended by induction if $p_i \neq p_j$ when $v_i \neq v_j$. Namely, $\forall\, T \leq R$, the following solution $\hat{\boldsymbol{z}}$ will be feasible:

$$\hat{\boldsymbol{z}} = \boldsymbol{z} + \sum_{i=1}^{T} \boldsymbol{e}_{v_i} - \sum_{i=1}^{T} \boldsymbol{e}_{p_i}. \quad (20)$$

If, for some $v_i \neq v_j$, we have $p_i = p_j$, then an even stronger statement holds: $\hat{\boldsymbol{z}} = \boldsymbol{z} - \boldsymbol{e}_{p_i} + \boldsymbol{e}_{v_i} + \boldsymbol{e}_{v_j}$ will be feasible (since subtracting $\boldsymbol{e}_{p_i}$ will "loosen" both constraints, instead of just one), and therefore $T = 1$ and $j = v_j$ satisfy (16).

So for the remaining proof of (16), we can restrict attention to the most general case of $v_i \neq v_j \Rightarrow p_i \neq p_j$. Let us define the following sets:

$$\hat{\mathcal{I}} = \{i \in \mathcal{O} \ : \ \hat{\boldsymbol{z}}_i = 1\}, \tag{21}$$

$$\beta = \left\{l \in \{1, \ldots, m\} \ : \ \exists\, i \in \hat{\mathcal{I}} \text{ s.t. } \boldsymbol{a}_l^{'}\, \boldsymbol{e}_i = 1\right\}, \tag{22}$$

$$\bar{\beta} = \{1, \ldots, m\} \setminus \beta. \tag{23}$$

$\hat{\mathcal{I}}$ is the set of all variables which are 1 in both $\hat{\boldsymbol{z}}$ and the optimal solution, $\boldsymbol{z}^*$. From the construction of $\hat{\boldsymbol{z}}$, it can be seen that $\hat{\mathcal{I}} = \mathcal{V} \cup (\mathcal{I} \cap \mathcal{O})$. From (17), $\forall\, v_i \in \mathcal{V} \Rightarrow v_i \notin \mathcal{I} \Rightarrow \mathcal{V} \cap \mathcal{I} = \emptyset \Rightarrow \left|\hat{\mathcal{I}}\right| = |\mathcal{V}| + |\mathcal{I} \cap \mathcal{O}|$. Letting $n_0 = |\mathcal{I} \cap \mathcal{O}|$, we have $\left|\hat{\mathcal{I}}\right| = R + n_0$.

β is the set of all constraints in which variables from $\hat{\mathcal{I}}$ participate. Since $\forall\, i \neq j \in \hat{\mathcal{I}} \Rightarrow i, j \in \mathcal{O}$, then, from (13), they cannot participate in the same constraint, so $|\beta| = (R + n_0)w$.

$\bar{\beta}$ is the set of all other constraints. $\left|\bar{\beta}\right| = m - w\,(R + n_0)$.

From (20), with $T = R$, we obtain that $\hat{\boldsymbol{z}} = \boldsymbol{z} + \sum_{i=1}^{R} \boldsymbol{e}_{v_i} - \sum_{i=1}^{R} \boldsymbol{e}_{p_i}$ is feasible. Since $\boldsymbol{e}'\hat{\boldsymbol{z}} = \boldsymbol{e}'\boldsymbol{z} < \frac{Z^*}{w}\left(2 - \frac{1}{w}\right)$, then, by an argument similar to (18), the number of tight constraints in $\hat{\boldsymbol{z}}$ is $< Z^*\left(2 - \frac{1}{w}\right)$. Furthermore, since $\hat{\boldsymbol{z}}_i = 1$, $\forall\, i \in \hat{\mathcal{I}}$, all the β constraints are tight, so the number of tight $\bar{\beta}$ constraints is $< Z^*\left(2 - \frac{1}{w}\right) - (R + n_0)w$. From (14), $R > Z^*/w \Rightarrow$

$$\begin{aligned} &Z^*\left(2 - \frac{1}{w}\right) - (R + n_0) \cdot w < Z^*\left(2 - \frac{1}{w}\right) - R \cdot w - n_0 \leq \\ &\leq Z^*\left(2 - \frac{1}{w}\right) - \frac{Z^*}{w}(w - 1) - R - n_0 = Z^* - R - n_0. \end{aligned} \tag{24}$$

Now consider all the variables in $\mathcal{O} \setminus \hat{\mathcal{I}}$. For any such variable j, $j \notin \hat{\mathcal{I}} \Rightarrow \hat{\boldsymbol{z}}_j = 0$ and j only participates in $\bar{\beta}$ constraints. Also, $\forall\ i \neq j \in \mathcal{O} \setminus \hat{\mathcal{I}}$, from (13), j and i cannot participate in the same constraint. But from (24), there are $< Z^* - R - n_0$ tight constraints involving variables j, and there are $|\mathcal{O}| - \left|\hat{\mathcal{I}}\right| = Z^* - R - n_0$ such j. Therefore $\exists\ j$ s.t. $\hat{\boldsymbol{z}} + \boldsymbol{e}_j$ is feasible, proving (16). □

The main result of the preceding lemma is (16), which indicates that for any solution $\boldsymbol{z}$ not satisfying the requirements of Theorem 2, a better feasible solution $\hat{\boldsymbol{z}} = \boldsymbol{z} + \sum_{i=1}^{R} \boldsymbol{e}_{v_i} - \sum_{i=1}^{R} \boldsymbol{e}_{p_i} + \boldsymbol{e}_j$ can be constructed, by:

1. Subtracting all the relevant $\boldsymbol{e}_{p_i}$

2. Adding $\boldsymbol{e}_j$

3. Adding all the corresponding $\boldsymbol{e}_{v_i}$

However, it is not immediately clear that our algorithm would proceed according to these steps. For instance, perhaps a solution $\boldsymbol{z} - \sum_{i=1}^{R} \boldsymbol{e}_{p_i}$ is never examined! As such, we need one more result concerning the reachability of $\hat{\boldsymbol{z}}$.

We introduce the concept of a *generation*, defined by the following recursion:

- Let the best feasible solution $\boldsymbol{z}$ always have generation 0.
- For any solution $\boldsymbol{y}$ inserted in the list at Step 12 of Algorithm 1, define its generation to be 1+ the generation of the solution $\boldsymbol{x}$ from Step 4, to which $\boldsymbol{y}$ is adjacent.

Observe that the definition is consistent: the generation counting is always reset when the current feasible solution $\boldsymbol{z}$ is updated in Step 6, since the solution list is cleared and $\boldsymbol{z}$, whose generation is set to 0, is the only solution added to the list. From that point onwards, for any solution $\boldsymbol{x}$ extracted and examined in Steps 3 and 4, the generation t will simply represent the number of variables that the algorithm has changed starting at $\boldsymbol{z}$ in order to reach $\boldsymbol{x}$. Note that this is not the same as the distance between $\boldsymbol{z}$ and $\boldsymbol{x}$. For instance, $\boldsymbol{x} = \boldsymbol{z} + \boldsymbol{e}_i - \boldsymbol{e}_i$ will actually be identical to $\boldsymbol{z}$, but it will have generation 2.

An immediate consequence of this assignment is that all the solutions $\boldsymbol{x}$ will be inserted into (and hence extracted from) the FIFO list in an increasing order of generations.

With variables $\boldsymbol{z}, R$ and indices p_i, v_i and j having the same significance as that from Lemma 3, we establish the following result:

Lemma 4. *If $Q \geq T \cdot w$ and $T \leq R$, a feasible solution of generation T with the same trace and objective function value as $\boldsymbol{z} - \sum_{i=1}^{T} \boldsymbol{e}_{p_i}$ will be in the FIFO list.*

Proof. First note that, as a consequence of (15), $\forall\, t \leq T,\ \boldsymbol{z} \geq \sum_{i=1}^{t} \boldsymbol{e}_{p_i}$, which makes the subtraction operations well defined. Furthermore, any such solution is feasible (since $\boldsymbol{z}$ itself is feasible), which also implies that any solution with the same trace as $\boldsymbol{z} \geq \sum_{i=1}^{t} \boldsymbol{e}_{p_i}$ must also be feasible.

The first step of the induction is trivial: generation 0 has $\boldsymbol{z}$ in the list. Assume that the property holds for solutions of the t-th generation, $t < T$, and call such a solution $\boldsymbol{z}^{(t)}$. Note that $\boldsymbol{z}^{(t)}$ is not necessarily equal to $\boldsymbol{z} - \sum_{i=1}^{t} \boldsymbol{e}_{p_i}$. It only has the same trace and objective function value.

We claim that $\boldsymbol{z}^{(t)}_{p_{t+1}} = 1$. Assume by contradiction that $\boldsymbol{z}^{(t)}_{p_{t+1}} = 0$. With any subtraction of $\boldsymbol{e}_{p_i}$, exactly w constraints become loose, and hence $\|trace(\boldsymbol{x}) - trace(\boldsymbol{z})\|_1$ increases by w. Thus the trace distance between $\boldsymbol{z}^{(t)}$ and $\boldsymbol{z}$ is exactly tw. If $\boldsymbol{z}^{(t)}_{p_{t+1}} = 0$, and the trace is the same as that of $\boldsymbol{z} - \sum_{i=1}^{t} \boldsymbol{e}_{p_i}$, then in some earlier generation, the variable at p_{t+1} was changed from 1 to 0. Also, to maintain the same trace, it must have been the case that one other variable was changed from 0 to 1 in each of the w constraints in which p_{t+1} participates. But this would cause a delay of at least 2 generations as compared to $\boldsymbol{z}^{(t)}$, meaning that such a solution could not have been already examined. It is the property of the FIFO list which imposes that solutions will be examined in a strictly increasing order of generations. Hence it must be that $\boldsymbol{z}^{(t)}_{p_{t+1}} = 1$.

But then, in the t-th generation, a solution with the same trace and objective function as $\boldsymbol{z}^{(t)} - \boldsymbol{e}_{p_{t+1}}$ will be examined. Since $Q \geq T \cdot w \geq (t+1)w$, and this solution is feasible, it will immediately satisfy Conditions (A1) and (A2) characterizing interesting solutions. For Condition (A3), there are two cases:

- If the objective function value for this solution is larger than that found in its corresponding trace box, the solution will be added to the list, with generation $t+1$ assigned to it, and the induction proof is complete.

- Otherwise, since all the trace boxes are set to $-\infty$ when the list is cleared, it must be that some other solution $\tilde{\boldsymbol{z}}$, mapping to the same trace box, was already added to the list in some earlier step. Since $h(\cdot)$ is injective, it must be that $trace(\tilde{\boldsymbol{z}}) = trace\left(\boldsymbol{z}^{(t)} - \boldsymbol{e}_{p_{t+1}}\right)$. But, as argued in the preceding paragraph, this would imply that the distance between $trace(\boldsymbol{z})$ and $trace(\tilde{\boldsymbol{z}})$ was exactly $(t+1)w$, meaning that at least $t+1$ variables were changed starting from $\boldsymbol{z}$ in order to reach $\tilde{\boldsymbol{z}}$. But then $\tilde{\boldsymbol{z}}$ must have generation $t+1$, completing our inductive proof. □

With the preceding lemmas, we are now ready to prove the result of Theorem 2.

Proof. Assume the heuristic is run with some initial feasible solution $\boldsymbol{z} = \boldsymbol{z}_0$ satisfying $\boldsymbol{e}'\boldsymbol{z} < \frac{2w-1}{w^2} Z^*$. If there are solutions $\boldsymbol{y}$ adjacent to $\boldsymbol{z}$ that satisfy the condition at Step 5 in the algorithm (namely, they are feasible and have better objective function value than $\boldsymbol{z}$), then Steps 6-8 will clear the solution list and replace $\boldsymbol{z}$ with $\boldsymbol{y}$. If repeating this process results in a feasible solution $\boldsymbol{z}$ satisfying equation (7), then there is nothing to prove. So, without loss of generality, let us assume that we reach a feasible solution $\boldsymbol{z}$ for which no

adjacent $\boldsymbol{y}$ satisfies the condition at Step 5. Then, from Lemma 3, a feasible solution $\hat{\boldsymbol{z}} = \boldsymbol{z} + \sum_{i=1}^{R} \boldsymbol{e}_{v_i} - \sum_{i=1}^{R} \boldsymbol{e}_{p_i} + \boldsymbol{e}_j$ exists.

By Lemma 4, after t generations, a solution $\boldsymbol{z}^{(t)}$ with the same trace and objective function value as $\boldsymbol{z} - \sum_{i=1}^{t} \boldsymbol{e}_{p_i}$ will be in the FIFO list. The number t of such generations that need to be considered is given by the first time when $\boldsymbol{e}_j$ can be added. Since e_j participates in w constraints, it will collide with at most w of the p_i, which must first be subtracted. Therefore, we require $t \geq w$, which, by Lemma 4, implies that $Q \geq w^2$, justifying the condition in the statement of the theorem.

Once all the p_i's are subtracted, in generation w, a feasible solution with the same trace and objective function as $\boldsymbol{z} - \sum_{i=1}^{w} \boldsymbol{e}_{p_i} + \boldsymbol{e}_j$ will be considered by the algorithm. By the same inductive argument as in the proof of Lemma 4, it can be seen that for all future generations $w + 1 + t$, a feasible solution with the same trace and objective function value as $\boldsymbol{z} - \sum_{i=1}^{w} \boldsymbol{e}_{p_i} + \boldsymbol{e}_j + \sum_{i=1}^{t} \boldsymbol{e}_{v_i}$ will be in the FIFO list. After $2w + 1$ generations, a feasible solution $\boldsymbol{z}^{(2w+1)} = \boldsymbol{z} - \sum_{i=1}^{w} \boldsymbol{e}_{p_i} + \boldsymbol{e}_j + \sum_{i=1}^{w} \boldsymbol{e}_{v_i}$, with objective function $\boldsymbol{e}'\boldsymbol{z}^{(2w+1)} = \boldsymbol{e}'\boldsymbol{z} + 1$, will be examined. In Step 6, the current feasible solution $\boldsymbol{z}$ will be replaced with $\boldsymbol{z}^{(2w+1)}$, and the solution list and trace boxes will be cleared.

Repeating this argument inductively for the new $\boldsymbol{z}$, we see that the end solution has to obey $\frac{Z^*}{Z_H} \leq \frac{w^2}{2w-1}$, proving Theorem 2. □

It is worth making the observation that this result is only slightly weaker than the best known bound of $\frac{w}{2} + \epsilon$ for w-set packing, found in (Hurkens and Schrijver, 1989). However, the latter bound is derived as a byproduct of a procedure applicable only to set packing, whereas our method pertains to general binary optimization problems.

5. Implementation

In this section, we present several details specific to our implementation of the algorithm. While these provide a guideline for several "good" choices of data-structures and parameter values, they should by no means be regarded as exhaustive or optimal. Our main reason for including them here is to provide a complete framework for the computational results in Section 6.

5.1. Problem Representation

In order to accommodate large data-sets, we have opted to implement the constraint matrix $\boldsymbol{A}$ as a sparse matrix (n sparse vectors, one for each column). The vectors $\boldsymbol{b} \in \mathbb{Z}^m$ and $\boldsymbol{c} \in \mathbb{Z}^n$ were represented as dense vectors (arrays of integer values).

For the solutions $\boldsymbol{x}$, we have taken two different approaches. In problems where $\boldsymbol{x} \in \mathbb{Z}^n$, solutions were represented as n-dimensional dense vectors. For problems with $\boldsymbol{x} \in \{0,1\}^n$, every solution was represented as a bit-array (also known as *bit-field* or *bitmap*). This compact representation significantly reduced the memory requirements, which turned out to be essential for achieving better performance.

Since the algorithm usually operated with *interesting* solutions, which, by Conditions (A1) and (A2) from Section 2.1, have few nonzero entries, we have decided to store the traces of solutions as sparse arrays.

5.2. Algorithm-characteristic Data Structures

As hinted to in earlier sections, the major choices in terms of implementation were the solution list $\mathcal{SL}$, with the associated trace boxes, and the function $h(\cdot)$.

Note that if no other restrictions are imposed, an implementation using a FIFO solution list, with a large Q, could create structures of very large size, since interesting solutions could be added for a very long time until a feasible solution of better objective function value is found, and the list is cleared. To fix this situation, we have decided to store as many solutions as there are trace boxes. After all, once a solution is deemed *interesting*, the previous solution mapping to the same trace box is no longer interesting, and hence could simply be ignored.

This brings us to the issue of the number of trace boxes. The ideal case of an injective $h(\cdot)$, which implies having one trace box for each possible trace of an interesting solution, would require $O\left(\binom{2m}{Q}\right)$ boxes, by equation (4). Since for every trace box, we would also like to store the associated interesting solution, this would imply a memory commitment of $O\left(n \cdot \binom{2m}{Q}\right)$, which for large m, n could cause problems even in modern systems.

As suggested in Section 2.1, one way to overcome these difficulties is to relax the requirement of having $h(\cdot)$ injective. Instead, we would consider a function $h : U \to V$, where $U \subset \{0,1\}^{2m}$ is the set of traces of interesting solutions and $V = \{1, 2, \ldots, N_{TB}\}$ is the set of indices of trace boxes. The parameter N_{TB} represents the total number of trace boxes that

can be considered, which is also the total size of the allowed solution list $\mathcal{SL}$. As such, it provides a direct connection with the total amount of memory committed to the algorithm, and can be adjusted depending on the available resources.

The advantage of this approach is that we are now free to choose N_{TB} and $h(\cdot)$. The main pitfall is that for most practical problems, $N_{TB} << |U|$, and hence multiple interesting solutions with different traces will map to the same trace box, causing some of them to be ignored in the search. If the number of such collisions is high, then the algorithm might ignore many "good" directions of improvement, resulting in poor performance. To minimize this undesirable effect, we take the following twofold approach:

1. We choose $h(\cdot)$ as a *hash function*, namely a mapping from a large universe of values (U) to a much smaller set (V), with as few collisions as possible.

2. Instead of having a single hash function $h(\cdot)$, i.e. allowing each trace of an interesting solution to map to a single trace box, we consider a family of hash functions $h^i(\cdot), i \in \{1, 2, \ldots, N_H\}$. The parameter N_H, representing the number of distinct trace boxes into which an interesting trace gets mapped, is a fixed, small number that becomes another choice in the design.

With the addition of multiple hash functions $h^i(\cdot)$, the original definition of an *interesting solution* from Section 2.1 has to be modified slightly. While the first two conditions remain the same, a solution $\boldsymbol{y}$ is now found interesting if $\boldsymbol{c}'\boldsymbol{y} > \boldsymbol{c}'\boldsymbol{x}$ for all $\boldsymbol{x}$ already examined such that $h^i\left(trace(\boldsymbol{x})\right) = h^i\left(trace(\boldsymbol{y})\right)$ for some $i \in \{1, \ldots, N_H\}$. In other words, in Step 10 of Algorithm 1, $\boldsymbol{y}$ is interesting if its objective function value $\boldsymbol{c}'\boldsymbol{y}$ is larger than at least one of the values stored in the trace boxes $h^1(trace(\boldsymbol{y}))$, $h^2(trace(\boldsymbol{y}))$, $\ldots$, $h^{N_H}(trace(\boldsymbol{y}))$. If that is the case, in Step 11, the value $\boldsymbol{c}'\boldsymbol{y}$ is stored in all the trace boxes satisfying this property, and the solution $\boldsymbol{y}$ is written in the corresponding locations in the solution list at Step 12.

The downside for using this approach is that the theoretical result presented in prior sections change for the worse. Namely, for a general cost vector $\boldsymbol{c} \in \mathbb{Z}^n$, with the number of trace boxes fixed to N_{TB} and the number of hash functions fixed to N_H, the running time from Section 3 becomes:

$$O\left(\|\boldsymbol{c}\|_1^{\,2} \cdot N_{TB} \cdot n \cdot max(m, n \cdot N_H)\right) \ , \tag{25}$$

and the performance guarantee from Section 4 is lost. However, as we will see in Section 6, this approach is advantageous from a computational perspective, and delivers very good results in practice.

5.3. Hash functions

To complete the description of the implementation, in this subsection we present our particular choice of functions $h^i(\cdot)$. While the literature on hash functions is abundant and many good choices are available (see Cormen et al. (2001) for an introduction and S. Bakhtiari and Pieprzyk (1995) for a survey article), we have settled for a less sophisticated version, which we describe in the next paragraphs.

In the first step, for each hash function $h^i,\ i \in \{1, 2, \ldots, N_H\}$, a set of m positive integer values was generated. These values were chosen uniformly at random, and only once, at the very beginning of the algorithm. Let the i-th set of such values be $\Phi^i = \{\phi^i_1,\ \phi^i_2,\ \ldots,\ \phi^i_m\}$.

Given the total (fixed) number N_{TB} of traces boxes, we distinguish the following two regions of equal size:

1. The first region, henceforth referred to as the "y_v region", corresponds to interesting solutions $\boldsymbol{y}$ with $\boldsymbol{y}_v \neq 0$ (i.e., violating certain constraints). This region is further split into subregions, depending on the number of violated constraints:

   - The first subregion, of size m, corresponds to solutions $\boldsymbol{y}$ with exactly one violated constraint ($\|\boldsymbol{y}_v\|_1 = 1$). Since there are m total constraints, the mapping into this region is trivial: a solution which violates only constraint i will be mapped to the i-th box of this region.
   - The remaining $(N_{TB}/2 - m)$ boxes from the y_v region are split evenly among exactly $Q - 1$ subregions. Any interesting solution $\boldsymbol{y}$, with violated constraints $j_1, j_2, \ldots, j_p$ $(2 \leq p \leq Q)$, would be mapped only to the p-th such subregion, and would have N_H boxes corresponding to it, one for each hash function. The i-th hash function would compute the corresponding trace box according to the following formula:

$$h^i\left[trace(\boldsymbol{y})\right] = \left(\sum_{k=1}^{p} \phi^i_{j_k} + \prod_{k=1}^{p} \phi^i_{j_k}\right) \mod \left(\frac{N_{TB}/2 - m}{Q - 1}\right),\quad i \in \{1, \ldots, N_H\} \tag{26}$$

Where ($a \mod b$) denotes the remainder obtained when dividing the integer a by the integer b. The above formula has a simple interpretation: the first term is a combination of the set Φ^i of random values, based on the indices $j_1, \ldots, j_p$ of the violated constraints. The *mod* operation ensures that the resulting index is in a range suitable for the p-th subregion. The intuition behind why the formula works and results in few collisions is more complicated, and is beyond the scope of the current paper (we refer the interested reader to (S. Bakhtiari and Pieprzyk, 1995) for a more comprehensive treatment).

2. The second region, also of size $N_{TB}/2$, corresponds to interesting solutions with no violated constraints ($\boldsymbol{y}_v = \mathbf{0}$), but with loose constraints ($\boldsymbol{y}_w \neq \mathbf{0}$). Similar to the previous discussion, this region is called the "y_w region", and is further divided into subregions:

   - The first subregion has size m, and corresponds to solutions with exactly one loose constraint. The mapping here is analogous to that from the y_v case.
   - The remaining $N_{TB}/2 - m$ boxes are divided evenly among the $Q - 1$ subregions corresponding to solutions with more than one loose constraint. However, unlike the situation with y_v, it is no longer desirable to map solutions with p loose constraints exclusively in the p-th subregion. Instead, these solutions should also be compared with solutions having fewer than p loose constraints. The intuition is that if a solution having more loose constraints also has higher objective function value, then it would be desirable to have it considered by the algorithm. To accommodate for this new provision, for each solution with loose constraints $j_1, \ldots, j_p$ ($p \geq 2$), we choose several subsets of 1, 2 or r constraints ($r \leq p$ could be either a function of p or chosen in some deterministic way). The numbers of such subsets, henceforth referred to as N_1, N_2 and N_r respectively, are fixed and become parameters of the algorithm. Furthermore, the choice of the subsets themselves is done in a deterministic fashion, so that for any particular trace of an interesting solution $\boldsymbol{y}$, the same subsets are always chosen. Once such a subset of indices $j_1, \ldots, j_r$ is fixed, the trace index is computed with the help of one of the hash functions defined before - for instance, we could use the very first hash

function:

$$h^1\left[trace(\boldsymbol{y})\right] = \left(\sum_{k=1}^{r} \phi_{j_k}^1 + \prod_{k=1}^{r} \phi_{j_k}^1\right) \mod \left(\frac{N_{TB}/2 - m}{Q-1}\right) \tag{27}$$

Note that since we are already considering multiple sets of indices, the same solution is automatically mapped into multiple boxes in the y_w region, so there is no need to compute the results from multiple hash functions, as was done for $\boldsymbol{y}_v$.

We conclude this section by making two relevant observations. First, note that since the "random" values Φ^i do not change during the run of the algorithm, the hash functions $h^i(\cdot)$ are deterministic, in the sense that the same trace of a particular solution $\boldsymbol{y}$ is always mapped to the same trace boxes, regardless of the time at which it is considered by the algorithm. Therefore, the set of rules specified above uniquely determines the way in which each interesting solution is mapped into the trace boxes (and, implicitly, in the solution list).

Second, observe that the number of trace boxes N_{TB} (or, equivalently, the total amount of memory committed to the solution list) and the parameter Q should, in general, not be chosen independently. The reason is that for a fixed N_{TB}, the size of each subregion in both the y_v and the y_w regions is inversely proportional with Q. Therefore, if we would like the parameter Q to be a good indicator of the performance of the algorithm (i.e., larger Q resulting in improved objective function value), then we should increase N_{TB} accordingly, so that the ratio N_{TB}/Q remains roughly constant.

5.4. Extracting a new solution from the list

The last relevant detail of the implementation is the way in which interesting solutions are extracted from the solution list $\mathcal{SL}$, at Step 3 of the algorithm. While any procedure that extracts solutions repeatedly would eventually explore all the interesting solutions, particular choices for the order of extraction could speed up the algorithm considerably. For example, it would be desirable to first examine solutions $\boldsymbol{y}$ adjacent to an interesting solution $\boldsymbol{x}$ that has very few violated constraints, because such directions are more likely to result in feasible solutions.

To this end, we have included in the implementation a simple scheme based on a *priority queue*. The main idea behind this data-type is that each element inserted in the queue also has an associated value, which determines its priority relative to the other elements in the queue. Whenever an extraction occurs, the first element to leave the queue is the one with

the highest priority among all the members of the queue. For a comprehensive treatment and other references, we refer the interested reader to (Cormen et al., 2001).

To implement this concept in our setting, whenever a solution $\boldsymbol{y}$ was determined as interesting at Step 10, a priority value $pv(\boldsymbol{y})$ was computed based on $\boldsymbol{y}$'s objective function value and the number of constraints it violated (we used a very simple, additive scheme). When $\boldsymbol{y}$ was written in the solution list at Step 12, the index of its corresponding trace box was introduced in the priority queue, with a priority of $pv(\boldsymbol{y})$. By following this rule, the solution $\boldsymbol{x}$ extracted at Step 3 always had the largest priority among all solutions present in the list.

The downside for using a priority queue is that we need to store an additional $O(N_{TB})$ values, and the complexity for inserting and/or extracting from the priority queue becomes $O(\log N_{TB})$, hence raising the overall complexity of the scheme. However, despite this seemingly higher computational load, the actual (physical) running time is usually decreased, since the heuristic spends less time searching in "infertile" directions.

5.5. Running the algorithm

We conclude this section by first summarizing the parameters that the user is free to choose in our implementation of the heuristic:

- Q - the parameter determining what constitutes an interesting solution.
- N_{TB} - the number of trace boxes, also equal to the size of the solution list. Since specifying a particular N_{TB} is equivalent to fixing a certain memory commitment (MEM) for the solution list, we have decided to use the latter for convenience.
- N_H - the number of hash functions, influencing how many boxes correspond to each interesting solution.
- N_1, N_2 and N_r - the number of subsets of 1, 2 or r loose constraints, respectively, which should be considered when computing the indices of the trace boxes.

In order to simplify the benchmarking of the algorithm, we have decided to fix some of the adjustable parameters to a choice that has consistently delivered good results in our experiments:

$$N_H = 2;\ N_1 = 2;\ N_2 = 2;\ N_r = 5.$$

With respect to the two remaining parameters, Q and MEM, we have found that the most natural way to run the heuristic procedure is in stages, by gradually increasing the values of both Q and MEM. The reason is that cold-starting the procedure directly with large values of Q and MEM would result in an unnecessarily large computational time spent in clearing the (large) solution list $\mathcal{SL}$, which is done whenever the current feasible solution is updated. Thus, to improve the physical running time, one should always first run the heuristic with smaller values of Q and MEM, which would (quickly) deliver better feasible solutions, that could in turn be used to warm-start the heuristic with larger Q and MEM.

6. Computational Results

We have tested our implementation of the algorithm on several classes of problems, and have compared the results with the output from CPLEX 9.1. All the tests were run on the Operations Research Center computational machine, which is a Dual Core Intel® Xeon® 5050 Processor (3.00GHz, 4MB Cache, 667MHz FSB), with 8GB of RAM (667MHz), running Ubuntu Linux.

Consistent with our remarks in the end of Section 5.5, we have used the values $N_H = 2, N_1 = 2, N_2 = 2, N_r = 5$, and the following sequence of runs of the heuristic in all the test-cases:

$$(1)\, Q = 4, MEM = 10\text{MB} \;\Rightarrow\; (2)\, Q = 4, MEM = 50\text{MB} \;\Rightarrow\; (3)\, Q = 6, MEM = 100\text{MB} \;\Rightarrow$$
$$(4)\, Q = 6, MEM = 250\text{MB} \;\Rightarrow\; (5)\, Q = 10, MEM = 1\text{GB} \;\Rightarrow\; (6)\, Q = 10, MEM = 2\text{GB} \;\Rightarrow$$
$$(7) \begin{cases} Q = 15, MEM = 6\text{GB} \\ Q = 20, MEM = 6\text{GB} \end{cases}$$

In step (7), the brace indicates that the two independent runs were both started with the same initial feasible solution, given by the output from the run in step (6). These two runs were still performed sequentially (i.e. *non-concurrently*), so that the total completion time of the heuristic was given by the sum of the completion times of stages (1) to (6) and the two runs in stage (7).

6.1. Set covering

The first type of problem that we considered was set covering:

$$\begin{aligned} \min \quad & \boldsymbol{c}'\boldsymbol{x} \\ \text{s.t.} \quad & \boldsymbol{A}\boldsymbol{x} \geq \boldsymbol{e} \\ & \boldsymbol{x} \in \{0,1\}^n, \quad \boldsymbol{A} \in \{0,1\}^{m \cdot n}. \end{aligned} \tag{28}$$

In order to have sufficiently large data-sets, we wrote a script that generated different instances of the problem. The script took as arguments the number of constraints (m), the number of variables (n) and the number of nonzero entries (w) in each column of $\boldsymbol{A}$. In addition, there were two parameters specifying lower and upper bounds on the entries in the cost vector $\boldsymbol{c}$, for the weighted version of the problem.

In the first class of tests, we considered $m = 1000$, $n = 2500$ and took $w = 3$, $w = 5$ or w random, between 3 and 7. Table 1 records the results of the simulation for the unweighted case, $\boldsymbol{c} = \boldsymbol{e}$, while Table 2 contains the results for the weighted case, where the weights c_i were also randomly generated, with values $c_i \in [400, 500]$.

Table 1: Results for Unweighted Set Covering. $m = 1000$, $n = 2500$, $\boldsymbol{c} = \boldsymbol{e}$.

time	5 hours		10 hours		20 hours	
$\boldsymbol{w}$	ALG1	CPLEX	ALG1	CPLEX	ALG1	CPLEX
3	344	344	344	344	344	344
5	231	229	231	229	231	229
$3 \div 7$	203	211	203	210	203	210

Table 2: Results for Weighted Set Covering. $m = 1000$, $n = 2500$, $c_i \in [400, 500]$.

time	5 hours		10 hours		20 hours	
$\boldsymbol{w}$	ALG1	CPLEX	ALG1	CPLEX	ALG1	CPLEX
3	150,219	151,363	149,978	151,363	149,744	150,843
5	100,261	103,426	99,858	103,426	99,532	103,426
$3 \div 7$	89,341	90,361	88,996	90,017	88,996	88,996

In the third category of tests, summarized in Table 3, we have considered a larger problem size, $m = 4,000$ and $n = 10,000$, with w random in $\{3, \ldots, 7\}$.

Based on the eight examples presented here and several other runs we have performed, our assessment is that Algorithm 1 outperforms CPLEX after approximately 20 hours (and in some cases earlier) when both methods are run with the same amount of memory (6 GB). For shorter running times (1-2 hours), CPLEX has an edge, although not in all cases.

Table 3: Results for large instance of Unweighted Set Covering. $m = 4000$, $n = 10000$, $w \in \{3, \ldots, 7\}$.

time	10 hours		20 hours		100 hours		200 hours	
remarks	ALG1	CPLEX	ALG1	CPLEX	ALG1	CPLEX	ALG1	CPLEX
unweighted	969	904	858	887	826	865	826	858
weighted	454,495	393,540	432,562	393,540	367,516	381,087	366,021	381,087

6.2. Set packing

The second problem we considered was set packing:

$$\begin{aligned} \max \quad & \boldsymbol{c}'\boldsymbol{x} \\ \text{s.t.} \quad & \boldsymbol{A}\boldsymbol{x} \leq \boldsymbol{e} \\ & \boldsymbol{x} \in \{0,1\}^n, \quad \boldsymbol{A} \in \{0,1\}^{m \cdot n}. \end{aligned} \tag{29}$$

In this case, we also used a script to generate the test cases. Just as with set covering, we took $m = 1000$, $n = 2500$ and $w = 3$, and ran both unweighted and weighted versions (for the latter, all the entries in $\boldsymbol{c}$ were generated randomly, with values between 400 and 500). The results for the two tests are recorded in Tables 4 and 5, respectively. Just as with set covering, we find that Algorithm 1 is able to outperform CPLEX after approximately 20 hours, when both methods are run with the same amount of memory (6 GB).

Table 4: Results for Unweighted Set Packing. $m = 1000$, $n = 2500$, $\boldsymbol{c} = \boldsymbol{e}$.

time	5 hours		10 hours		20 hours	
$\boldsymbol{w}$	ALG1	CPLEX	ALG1	CPLEX	ALG1	CPLEX
3	319	317	320	317	320	317
5	164	158	167	158	167	160
$3 \div 7$	236	237	239	237	239	238

Table 5: Results for Weighted Set Packing. $m = 1000$, $n = 2500$, $c_i \in [400, 500]$.

time	5 hours		10 hours		20 hours	
$\boldsymbol{w}$	ALG1	CPLEX	ALG1	CPLEX	ALG1	CPLEX
3	146,984	146,771	146,984	147,877	147,604	147,877
5	76,258	71,782	76,258	72,592	77,086	72,592
$3 \div 7$	108,077	107,447	108,722	107,447	109,264	107,463

7. Conclusions

In this paper, we have presented a new class of general-purpose heuristic methods for solving large, sparse binary optimization problems. The formulation of the central algorithm, based on the notion of *interesting* solutions and their *traces*, provides flexibility in terms of the exact implementation, and allows the user to directly influence the complexity-performance tradeoff through the adjustable parameter Q.

In addition to interesting theoretical properties (pseudo-polynomial running times and performance guarantees), we feel that the proposed method has practical promise, as it is generally applicable, and it is either competitive with or strictly better than the leading optimization package in preliminary computational tests of fairly large instances of randomly generated binary optimization problems.

8. Appendix

The interested reader can find below the continuation of the set-packing example (2) from Section 2, which we restate for ease of exposition:

$$\begin{array}{lrrrl} \max & x_1 & +x_2 & +x_3 & \\ \text{s.t.} & x_1 & & +x_3 & \leq 1 \\ & & x_2 & +x_3 & \leq 1 \\ & \multicolumn{4}{c}{x_1, x_2, x_3 \in \{0,1\}} \end{array}$$

- (Steps taken in Section 2, ending with the following values for the variables:

$$\boldsymbol{z} = [1,1,0];\ \mathcal{SL} = \{\,[1,1,0]\,\};\ \mathcal{TB}[i] = -\infty, \forall\, i. \quad)$$

- (Step 3) $\boldsymbol{x} \leftarrow [1,1,0]$. Adjacent solutions are $[0,1,0], [1,0,0], [1,1,1]$.
  - (Step 4) $\boldsymbol{y} = [0,1,0]$, $trace(\boldsymbol{y}) = [0,0;1,0]$.
    - (Step 5) $\boldsymbol{y}$ feasible, but $\boldsymbol{e}'\boldsymbol{y} = 1 < \boldsymbol{e}'\boldsymbol{z}\,(=2)$.
    - (Step 10) (A1)-(A3) true, so $\boldsymbol{y}$ is interesting.
    - (Steps 11 - 12) $\mathcal{TB}[2] \leftarrow 1$; $\mathcal{SL} \leftarrow \{\,[0,1,0]\,\}$.
  - (Step 4) $\boldsymbol{y} = [1,0,0]$, $trace(\boldsymbol{y}) = [0,0;0,1]$.
    - (Step 5) $\boldsymbol{y}$ feasible, but $\boldsymbol{e}'\boldsymbol{y} = 1 < \boldsymbol{e}'\boldsymbol{z}\,(=2)$.
    - (Step 10) (A1)-(A3) true, so $\boldsymbol{y}$ is interesting.

* (Steps 11 - 12) $\mathcal{TB}[1] \leftarrow 1$; $\mathcal{SL} \leftarrow \{\,[0,1,0];[1,0,0]\,\}$.

– (Step 4) $\boldsymbol{y} = [1,1,1]$, $trace(\boldsymbol{y}) = [1,1;0,0]$.

* (Step 5) $\boldsymbol{y}$ infeasible.

* (Step 10) (A2) false, since $\|trace(\boldsymbol{y}) - trace(\boldsymbol{z})\|_1 = \|[1,1;0,0]\|_1 = 2 > Q$, so $\boldsymbol{y}$ is not interesting.

- (Step 2) $\mathcal{SL} = \{\,[0,1,0];[1,0,0]\,\} \neq \emptyset$.

- (Step 3) $\boldsymbol{x} \leftarrow [0,1,0]$. Adjacent solutions are $[1,1,0], [0,0,0], [0,1,1]$.

– (Step 4) $\boldsymbol{y} = [1,1,0]$, $trace(\boldsymbol{y}) = [0,0;0,0]$.

* (Step 5) $\boldsymbol{y}$ feasible, but $\boldsymbol{e}'\boldsymbol{y} = 2 = \boldsymbol{e}'\boldsymbol{z}$.

* (Step 10) (A1)-(A3) true, so $\boldsymbol{y}$ is interesting.

* (Steps 11 - 12) $\mathcal{TB}[0] \leftarrow 2$; $\mathcal{SL} \leftarrow \{\,[1,0,0];[1,1,0]\,\}$.

– (Step 4) $\boldsymbol{y} = [0,0,0]$, $trace(\boldsymbol{y}) = [0,0;1,1]$.

* (Step 5) $\boldsymbol{y}$ feasible, but $\boldsymbol{e}'\boldsymbol{y} = 0 < \boldsymbol{e}'\boldsymbol{z}\,(= 2)$.

* (Step 10) (A2) false, since $\|trace(\boldsymbol{y}) - trace(\boldsymbol{z})\|_1 = \|[0,0;1,1]\|_1 = 2 > Q$, so $\boldsymbol{y}$ is not interesting.

– (Step 4) $\boldsymbol{y} = [0,1,1]$, $trace(\boldsymbol{y}) = [0,1;0,0]$.

* (Step 5) $\boldsymbol{y}$ infeasible.

* (Step 10) (A1)-(A3) true, so $\boldsymbol{y}$ is interesting.

* (Steps 11 - 12) $\mathcal{TB}[4] \leftarrow 2$; $\mathcal{SL} \leftarrow \{\,[1,0,0];[1,1,0];[0,1,1]\,\}$.

- (Step 2) $\mathcal{SL} = \{\,[1,0,0];[1,1,0];[0,1,1]\,\} \neq \emptyset$.

- (Step 3) $\boldsymbol{x} \leftarrow [1,0,0]$. Adjacent solutions are $[0,0,0], [1,1,0], [1,0,1]$.

– (Step 4) $\boldsymbol{y} = [0,0,0]$, $trace(\boldsymbol{y}) = [0,0;1,1]$.

* (Step 5) $\boldsymbol{y}$ feasible, but $\boldsymbol{e}'\boldsymbol{y} = 0 < \boldsymbol{e}'\boldsymbol{z}\,(= 2)$.

* (Step 10) (A2) false, since $\|trace(\boldsymbol{y}) - trace(\boldsymbol{z})\|_1 = \|[0,0;1,1]\|_1 = 2 > Q$, so $\boldsymbol{y}$ is not interesting.

– (Step 4) $\boldsymbol{y} = [1,1,0]$, $trace(\boldsymbol{y}) = [0,0;0,0]$.

* (Step 5) $\boldsymbol{y}$ feasible, but $\boldsymbol{e}'\boldsymbol{y} = \boldsymbol{e}'\boldsymbol{z} = 2$.

        * (Step 10) (A3) false, since $\boldsymbol{e}'\boldsymbol{y} = \mathcal{TB}[0] = 2$, so $\boldsymbol{y}$ is not interesting.
    - (Step 4) $\boldsymbol{y} = [1, 0, 1]$, $trace(\boldsymbol{y}) = [1, 0; 0, 0]$.
        * (Step 5) $\boldsymbol{y}$ infeasible.
        * (Step 10) (A1)-(A3) true, so $\boldsymbol{y}$ is interesting.
        * (Steps 11 - 12) $\mathcal{TB}[8] \leftarrow 2$; $\mathcal{SL} \leftarrow \{\, [1, 1, 0]; [0, 1, 1]; [1, 0, 1] \,\}$.
- (Step 2) $\mathcal{SL} = \{\, [1, 1, 0]; [0, 1, 1]; [1, 0, 1] \,\} \neq \emptyset$.
- (Step 3) $\boldsymbol{x} \leftarrow [1, 1, 0]$. Adjacent solutions are $[0, 1, 0], [1, 0, 0], [1, 1, 1]$.
    - (Step 4) $\boldsymbol{y} = [0, 1, 0]$, $trace(\boldsymbol{y}) = [0, 0; 1, 0]$.
        * (Step 5) $\boldsymbol{y}$ feasible, but $\boldsymbol{e}'\boldsymbol{y} = 1 < \boldsymbol{e}'\boldsymbol{z}\,(= 2)$.
        * (Step 10) (A3) false, since $\boldsymbol{e}'\boldsymbol{y} = 1 = \mathcal{TB}[2]$, so $\boldsymbol{y}$ is not interesting.
    - (Step 4) $\boldsymbol{y} = [1, 0, 0]$, $trace(\boldsymbol{y}) = [0, 0; 0, 1]$.
        * (Step 5) $\boldsymbol{y}$ feasible, but $\boldsymbol{e}'\boldsymbol{y} = 1 < \boldsymbol{e}'\boldsymbol{z}\,(= 2)$.
        * (Step 10) (A3) false, since $\boldsymbol{e}'\boldsymbol{y} = \mathcal{TB}[1] = 1$, so $\boldsymbol{y}$ is not interesting.
    - (Step 4) $\boldsymbol{y} = [1, 1, 1]$, $trace(\boldsymbol{y}) = [1, 1; 0, 0]$.
        * (Step 5) $\boldsymbol{y}$ infeasible.
        * (Step 10) (A2) false, so $\boldsymbol{y}$ is not interesting.
- (Step 2) $\mathcal{SL} = \{\, [0, 1, 1]; [1, 0, 1] \,\} \neq \emptyset$.
- (Step 3) $\boldsymbol{x} \leftarrow [0, 1, 1]$. Adjacent solutions are $[1, 1, 1], [0, 0, 1], [0, 1, 0]$.
    - (Step 4) $\boldsymbol{y} = [1, 1, 1]$. Infeasible, and not interesting.
    - (Step 4) $\boldsymbol{y} = [0, 0, 1]$, $trace(\boldsymbol{y}) = [0, 0; 0, 0]$.
        * (Step 5) $\boldsymbol{y}$ feasible, but $\boldsymbol{e}'\boldsymbol{y} = 1 < \boldsymbol{e}'\boldsymbol{z}\,(= 2)$.
        * (Step 10) (A3) false, since $\boldsymbol{e}'\boldsymbol{y} = 1 < \mathcal{TB}[0] = 2$, so $\boldsymbol{y}$ is not interesting.
    - (Step 4) $\boldsymbol{y} = [0, 1, 0]$. Feasible, but not better than $\boldsymbol{z}$. Not interesting, since (A3) false.
- (Step 2) $\mathcal{SL} = \{\, [1, 0, 1] \,\} \neq \emptyset$.
- (Step 3) $\boldsymbol{x} \leftarrow [1, 0, 1]$. Adjacent solutions are $[0, 0, 1], [1, 1, 1], [1, 0, 0]$.

  - (Step 4) $\boldsymbol{y} = [0, 0, 1]$. Feasible, but not better than $\boldsymbol{z}$. Not interesting, since (A3) false.
  - (Step 4) $\boldsymbol{y} = [1, 1, 1]$. Infeasible. Not interesting, since (A2) false.
  - (Step 4) $\boldsymbol{y} = [1, 0, 0]$. Feasible, but not better than $\boldsymbol{z}$. Not interesting, since (A3) false.

- (Step 2) $\mathcal{SL} = \emptyset$.
- (Step 13) Return $z = [1, 1, 0]$.

References

Aarts, Emile, Jan K. Lenstra, eds. 1997. *Local Search in Combinatorial Optimization*. John Wiley & Sons, Inc., New York, NY, USA.

Balas, Egon, Sebastián Ceria, Gérard Cornuéjols. 1993. A lift-and-project cutting plane algorithm for mixed 0-1 programs. *Math. Program.* **58** 295–324.

Balas, Egon, Clarence. H. Martin. 1980. Pivot and complement-a heuristic for 0-1 programming. *Management Science* **26** 86–96.

Cormen, Thomas H., Clifford Stein, Ronald L. Rivest, Charles E. Leiserson. 2001. *Introduction to Algorithms*. McGraw-Hill Higher Education.

Eckstein, Jonathan, Mikhail Nediak. 2007. Pivot, cut, and dive: a heuristic for 0-1 mixed integer programming. *Journal of Heuristics* **13** 471–503.

Fischetti, Matteo, Fred Glover, Andrea Lodi. 2005. The feasibility pump. *Math. Program.* **104** 91–104.

Hurkens, C. A. J., A. Schrijver. 1989. On the size of systems of sets every t of which have an sdr, with an application to the worst-case ratio of heuristics for packing problems. *SIAM J. Discret. Math.* **2** 68–72.

S. Bakhtiari, R. Safavi-Naini, J. Pieprzyk. 1995. Cryptographic hash functions: A survey. Tech. Rep. 95-09, University of Wollongong - Department of Computer Science.