# Computational Experiments for Local Search Algorithms for Binary and Mixed Integer Optimization



by

Jingting Zhou

B.E. Biomedical Engineering, Zhejiang University, 2009

Submitted to the School of Engineering
in partial fulfillment of the requirements for the degree of

Master of Science in Computation for Design and Optimization

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 2010



Author ..........................................................
School of Engineering
August 4, 2010

Certified by ..........................................................
Dimitris J. Bertsimas
Boeing Professor of Operations Research
Thesis Supervisor

Accepted by ..........................................................
Karen Willcox
Associate Professor of Aeronautics and Astronautics
Codirector, Computation for Design and Optimization Program

# Computational Experiments for Local Search Algorithms for Binary and Mixed Integer Optimization

by

Jingting Zhou

Submitted to the School of Engineering
on August 4, 2010, in partial fulfillment of the
requirements for the degree of
Master of Science in Computation for Design and Optimization

## Abstract

In this thesis, we implement and test two algorithms for binary optimization and mixed integer optimization, respectively. We fine tune the parameters of these two algorithms and achieve satisfactory performance. We also compare our algorithms with CPLEX on large amount of fairly large-size instances. Based on the experimental results, our binary optimization algorithm delivers performance that is strictly better than CPLEX on instances with moderately dense constraint matrices, while for sparse instances, our algorithm delivers performance that is comparable to CPLEX. Our mixed integer optimization algorithm outperforms CPLEX most of the time when the constraint matrices are moderately dense, while for sparse instances, it yields results that are close to CPLEX, and the largest gap relative to the result given by CPLEX is around 5%. Our findings show that these two algorithms, especially the binary optimization algorithm, have practical promise in solving large, dense instances of both set covering and set packing problems.

Thesis Supervisor: Dimitris J. Bertsimas
Title: Boeing Professor of Operations Research

## Acknowledgments

After hundreds of hours' effort, I have fulfilled my thesis at MIT. It is an absolutely tough and challenging task. And without the following people for their help, I couldn't achieve my goal eventually.

First and foremost, I would like to express my deepest gratitude to my supervisor, Professor Dimitris Bertsimas, for his invaluable guidance through my thesis. I expecially appreciate the time he devoted to guide me, his insightful opinions, and his encouragement to cheer me up in face of difficulties. He has also set a role model to me, for his endless passion on work and great dedication to operations research.

My gratitude extends to Dan Iancu and Vineet Goyal, who have given me enormous support in my project. They are always very responsive to my inquiries and willing to spend time with me discussing my project. I am also touched by the friendliness of people at ORC, Andy, Michael and Bill, without their help, I couldn't keep my project going so smoothly.

I have enjoyed a great time at MIT for the past year, all because of you, my beloved SMA friends. The trip to Orlando, the gatherings at Mulan, we have spent so many fun moments together. I have also enjoyed so much the chat with Gil, Jamin, Wombi and Joel. They have shown great support to my work as the representative of CDO at GSC.

I would also like to thank SMA for providing a fellowship to support my study at MIT, and the staff at the MIT CDO office and SMA office who have made this period in Boston one of the greatest memory in my life.

Last but not least, I owe my deepest thanks to my parents for their unconditional love and belief in me. My love for them is more than words that I can say.

# Contents

# List of Tables

# Chapter 1

# Introduction

In the real world of optimization, binary optimization problems and mixed integer optimization problems have wide applications. Thus, extensive attention has been put into developing efficient algorithms over the past few decades, and considerable progress in our ability to solve those problems has been made. The emergence of major commercial codes such as CPLEX and EXPRESS is testimony to this fact, as they are able to solve such large scale problems. While part of the success can be attributed to significant speedups in computing power, there are two major elements that lead to the algorithmic development (see Aarts and Lenstra, 1997 [1] for a review): one is the introduction of new cutting plane methods (Balas et al., 1993) [2]; another is the use of heuristic algorithms, including the pivot-and-complement heuristic (Balas and Martin, 1980) [3], the "feasibility pump" (Fischetti et al., 2005) [5], and the pivot-cut-dive heuristic (Eckstein and Nediak, 2007) [4].

Despite the considerable progress in the field, we still have difficulty in solving especially dense binary problems and mixed integer problems. In addition, there is strong demand in the real-world applications to find better feasible solutions, without necessarily proving their optimality. In this thesis, we test two algorithms: one is a general purpose local search algorithm for binary optimization proposed by Bertsimas, Iancu and Katz [7], and the other is an adaptive local search algorithm for solving mixed integer optimization problems proposed by Bertsimas and Goyal[8]. In this thesis, we provide empirical evidence for their strength. Specifically, our contributions are as follows:

1. We implement those two algorithms. Furthermore, we propose a warm start sequence to reconcile the trade-off between algorithmic performance and complexity. In addition, we perform computational experiments to investigate the implementation details that affect algorithmic performance.

2. Most importantly, we compare the performance of these two algorithms with CPLEX on different types of fairly large instances, including the set covering and set packing instances for the binary optimization problem, and the set packing instances for the mixed integer optimization problem, with very encouraging results. Specifically, while the mixed integer optimization algorithm is comparable to CPLEX on moderately dense instances, the binary optimization algorithm strictly outperforms CPLEX on dense instances after 5 hours, 10 hours and 20 hours and is competitive with CPLEX on sparse instances.

The structure of rest of the thesis is as follows. In Chapter 2, we explain the binary optimization algorithm, discuss its implementation details, elaborate our experimental design, introduce several modified versions of the algorithm, present and analyze the computational results. In Chapter 3, we present the mixed integer optimization algorithm, discuss its implementation details, elaborate our experimental design and parameter choices, present and analyze the computational results.

# Chapter 2

# A General Purpose Local Search Algorithm for Binary Optimization

## 2.1 Problem Definition

The general problem is defined as a minimization problem with binary variables. The cost vector, constraint coefficients, and the right hand side (RHS), denoted as $\boldsymbol{c}, \boldsymbol{A}$, and $\boldsymbol{b}$, take integer values. This problem is referred to as the binary optimization problems (IP).

$$
\begin{aligned}
\min\ & \boldsymbol{c}^T \boldsymbol{x} \\
\text{s.t.}\ & \boldsymbol{A}\boldsymbol{x} \geq \boldsymbol{b} \\
& \boldsymbol{x} \in \{0,1\}^n,
\end{aligned}
\tag{2.1}
$$

where $\boldsymbol{A} \in \mathbb{Z}^{m \times n}$, $\boldsymbol{b} \in \mathbb{Z}^m$, $\boldsymbol{c} \in \mathbb{Z}^n$.

An important special case is the set covering problem. The constraint coefficients take binary values, and the RHS are all ones.

$$\begin{aligned} \min\ & \boldsymbol{c}^T\boldsymbol{x} \\ \text{s.t. } & \boldsymbol{A}\boldsymbol{x} \geq \boldsymbol{e} \\ & \boldsymbol{x} \in \{0,1\}^n, \end{aligned} \tag{2.2}$$

where $\boldsymbol{A} \in \{0,1\}^{m\times n}$, $\boldsymbol{c} \in \mathbb{Z}^n$.

We also test the algorithm's performance on the set packing problem, which is a maximization problem with binary variables. The constraint coefficients take binary values, and the RHS are all ones.

$$\begin{aligned} \max\ & \boldsymbol{c}^T\boldsymbol{x} \\ \text{s.t. } & \boldsymbol{A}\boldsymbol{x} \leq \boldsymbol{e} \\ & \boldsymbol{x} \in \{0,1\}^n, \end{aligned} \tag{2.3}$$

where $\boldsymbol{A} \in \{0,1\}^{m\times n}$, $\boldsymbol{c} \in \mathbb{Z}^n$.

Problem 2.3 can be converted into a minimization problem:

$$\begin{aligned} -(\min\ & -\boldsymbol{c}^T\boldsymbol{x}) \\ \text{s.t. } & -\boldsymbol{A}\boldsymbol{x} \geq -\boldsymbol{e} \\ & \boldsymbol{x} \in \{0,1\}^n, \end{aligned} \tag{2.4}$$

where $\boldsymbol{A} \in \{0,1\}^{m\times n}$, $\boldsymbol{c} \in \mathbb{Z}^n$.

We can solve the above problem 2.4 using the same algorithm as the set covering problem. The initial solution $\boldsymbol{x}$ is set to be all zeros instead of all ones in the set packing problem. The algorithm takes as input the matrix $-\boldsymbol{A}$, the vectors $-\boldsymbol{e}$ and $-\boldsymbol{c}$, the calculated objective function value is the opposite of the true objective.

## 2.2 Algorithm

We test the binary optimization algorithm proposed by Bertsimas, Iancu and Katz [7]. The algorithm, denotes as $\mathbb{B}$, takes as inputs the matrix $\boldsymbol{A}$, the vectors $\boldsymbol{b}$ and $\boldsymbol{c}$, the parameters $Q$ and $MEM$, and an initial solution $\boldsymbol{z}_0$. It generates feasible solutions with monotonically decreasing objective during the search process. The parameter $Q$ controls the search depth of the searching neighborhood, which poses a tradeoff between solution quality and computational complexity. The parameter $MEM$ controls the solution list size, which affects the degree of collision of interesting solutions with similar characteristics in constraint violation and looseness.

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

- $V(\boldsymbol{x}) = \max(\boldsymbol{b} - \boldsymbol{A}\boldsymbol{x}, \boldsymbol{0}) \in \mathbb{Z}_+^m$: the amount of constraint violation produced by $\boldsymbol{x}$.
- $U(\boldsymbol{x}) = \max(\boldsymbol{A}\boldsymbol{x} - \boldsymbol{b}, \boldsymbol{0}) \in \mathbb{Z}_+^m$: the amount of constraint looseness produced by $\boldsymbol{x}$.
- $W(\boldsymbol{x}) = \min(U(\boldsymbol{x}), \boldsymbol{e}) \in \{0,1\}^m$
- $trace(\boldsymbol{x}) = [V(\boldsymbol{x}); W(\boldsymbol{x}) - W(\boldsymbol{z})] \in \mathbb{Z}_+^m \times \{0,1\}^m$, where $\boldsymbol{z}$ is the current best feasible solution at a certain iteration of the algorithm.

Further, we introduce the following concepts:

- Two solutions $\boldsymbol{x}$ and $\boldsymbol{y}$ are said to be *adjacent* if $\mathbf{e}^T|\boldsymbol{x} - \boldsymbol{y}| = 1$.
- A feasible solution $\boldsymbol{z}_1$ is said to be *better* than another feasible solution $\boldsymbol{z}_2$ if $\mathbf{c}^T\boldsymbol{z}_1 < \mathbf{c}^T\boldsymbol{z}_2$.
- A solution $\boldsymbol{y}$ is said to be *interesting* if the following three criteria hold:

  **(A1)** $\|V(\boldsymbol{y})\|_\infty \leq 1$: no constraint is violated by more than one unit. If $\boldsymbol{A} \in \mathbb{Z}_+^{m \times n}$, we need to adjust this criterion to $\|V(\boldsymbol{y})\|_\infty \leq C$, $C$ is a constant that reflects the tolerance on the largest amount of violation. In this case, if we change an entry in $\boldsymbol{x}$ by one unit, the resulting violation may exceed one unit.

**(A2)** The number of violated constraints incurred by $\boldsymbol{y}$ is at most $Q$.

**(A3)** $\boldsymbol{c}^T\boldsymbol{y} < \boldsymbol{c}^T\boldsymbol{x}, \forall \boldsymbol{x}$ already examined by the algorithm, satisfying $h(trace(\boldsymbol{x})) = h(trace(\boldsymbol{y}))$. Here, $h : \{0,1\}^{2m} \rightarrow \mathbb{N}$ is a linear function that maps a vector into an integer. The mapping is multiple-to-one, that is, different vectors may be mapped to the same integer. The specifications of $h(\cdot)$ are elaborated in Section 2.3.

- A solution list $SL$

  All of the interesting solutions are stored in the solution list and ordered according to their assigned priority values. The priority value is computed based on the objective of the solution and the number of violations it incurred using a simple additive scheme. The solution list is maintained as a heap, thus the solution with the highest priority is extracted first. More detailed explanation is in Section 2.3.

- A trace box $TB$

  The trace box entry $TB[i]$ stores the best objective of an interesting solution $\boldsymbol{x}$ satisfying $h(trace(\boldsymbol{x})) = i$.

- The number of trace boxes $N_{TB}$

  There are $O\left(\begin{pmatrix} 2m \\ Q \end{pmatrix}\right)$ different traces for an injective $h(\cdot)$, which means one trace box for each possible trace. In this case, we need a memory commitment of $O\left(n \cdot \begin{pmatrix} 2m \\ Q \end{pmatrix}\right)$ for the solution list. For problems with large $m$ and $n$, this would cause difficulty in memory allocation. Here, we consider a function $h : U \rightarrow V$, where $U \subset \{0,1\}^{2m}$ is the set of traces of interesting solutions and $V = \{1, 2, ...N_{TB}\}$ is the set of indices of trace boxes. By choosing $N_{TB}$ and $h(\cdot)$, multiple interesting solutions with different traces may be mapped to the same trace box. This will inevitably lead to collision of interesting solutions and some of them will be ignored in the search. If such collision is high, the algorithm may perform poorly because it ignores many good directions. To minimize this undesirable effect, in our algorithm, we choose $h(\cdot)$ to be a hash function with small number of collisions and consider the following family of hash functions $h^i(\cdot), i \in \{1, 2, \ldots, N_H\}$. The parameter $N_H$ denotes the number of distinct trace boxes a trace will be mapped to. And the criterion of judging whether a solution is interesting also slightly changes. A solution $\boldsymbol{y}$ is interesting if its objective, $\boldsymbol{c}^T\boldsymbol{y}$, is larger than at least one of the values stored in the trace boxes $h^1(trace(\boldsymbol{y})), h^2(trace(\boldsymbol{y})), \ldots, h^{N_H}(trace(\boldsymbol{y}))$. For those trace boxes where solution $\boldsymbol{y}$ has a better objective, their values and corresponding solution in the $SL$ is updated to $\boldsymbol{c}^T\boldsymbol{y}$ and $\boldsymbol{y}$.

- The memory commitment for the solution list $MEM$

  Each trace box corresponds to an entry in the solution list, which stores the solution $\boldsymbol{x}$. Thus, the number of entries in the solution list $SL$ is the same as the number of trace boxes $N_{TB}$. The number of trace boxes $N_{TB}$ times the memory to store a solution $\boldsymbol{x}$ equals to the memory commitment for the solution list $SL$. Hereafter, the parameter $MEM$ refers to the allocated memory for solution list, which is equivalent to specifying a particular $N_{TB}$.

In our implementation, we change the following specifications of the algorithm compared to the original one in paper [7]:

- For condition (A1) of an interesting solution, instead of examining $\|trace(\boldsymbol{y}) - trace(\boldsymbol{z})\|_1 \leq Q$, we examine whether the number of violated constraints incurred by $\boldsymbol{y}$ is at most $Q$, which ignored the relative amount of looseness constraints.

- Instead of calculating the trace as $trace(\boldsymbol{x}) = [V(\boldsymbol{x}); W(\boldsymbol{x})] \in \mathbb{Z}_+^m \times \{0,1\}^m$, $trace(\boldsymbol{x})$ is calculated as follows: $trace(\boldsymbol{x}) = [V(\boldsymbol{x}); W(\boldsymbol{x}) - W(\boldsymbol{z})] \in \mathbb{Z}_+^m \times \{0,1\}^m$, let $\boldsymbol{z}$ to be the current best feasible solution.

More specifically, we give an outline of the algorithm as follows.

Input: matrix $\boldsymbol{A}$; vectors $\boldsymbol{b}$, $\boldsymbol{c}$; feasible solution $\boldsymbol{z}_0$; scalar parameters $Q, MEM$

Output: Best feasible solution $\boldsymbol{z}$

1. $\boldsymbol{x} = \boldsymbol{z}_0$; $SL = \boldsymbol{x}$ [$MEM$ is specified to determine the size of the $SL$]

2. while ($SL \neq \emptyset$)

3. get a new solution $\boldsymbol{x}$ from $SL$

4. for each ($\boldsymbol{y}$ adjacent to $\boldsymbol{x}$)

5. if ($\boldsymbol{A}\boldsymbol{y} \geq \boldsymbol{b}$) & ($\boldsymbol{c}^T\boldsymbol{y} \leq \boldsymbol{c}^T\boldsymbol{z}$)

6. $\boldsymbol{z} \leftarrow \boldsymbol{y}$

7. $SL \leftarrow \emptyset$

8. $SL \leftarrow SL \cup \boldsymbol{y}$

9. go to step 3

10. else if ($\boldsymbol{y}$ is interesting [$Q$ is specified for condition **A2**])

11. $TB[h(trace(\boldsymbol{y})] \leftarrow \boldsymbol{c}^T\boldsymbol{y}$

12. $SL \leftarrow SL \cup \boldsymbol{y}$

13. return $\boldsymbol{z}$

The algorithm starts with an initial feasible solution. In a typical iteration, the algorithm will select an interesting solution $\boldsymbol{x}$ from the solution list $SL$ and examine all its adjacent solutions. For each adjacent solution, $\boldsymbol{y}$, that is interesting (refer to the definition of interesting solutions in Section 2.2), we store it in the solution list and update the appropriate trace boxes. If we find a better feasible solution in this process, we clear the solution list and trace boxes, and jump to solution $\boldsymbol{z}$. The previous procedure resumes by examining the adjacent solutions of $\boldsymbol{z}$.

## 2.3 Implementation Details

In order to utilize the memory efficiently, we use the following data structures to represent the problem and the solution.

- We store the matrix $\boldsymbol{A}$ and vectors $V(\boldsymbol{x}), U(\boldsymbol{x}), W(\boldsymbol{x})$ as a sparse matrix and sparse vectors respectively, i.e., only store the indices and values of nonzero entries.

- We store the solution $\boldsymbol{x}$ in binary representation, which decreases the storage commitment for a solution $\boldsymbol{x}$ from $n$ to $\frac{n}{\text{sizeof}(int)} + 1$ integers.

We use hash function to map a solution's trace into an integer index. As mentioned before, we choose multiple hash functions and therefore, a trace may be mapped to multiple indices. We divide the trace boxes into two regions. We first examine the violation vector and we only examine the looseness vector when there is no violation. Given the fixed number of trace boxes $N_{TB}$, we define the following two regions of equal size $N_{TB}/2$:

1. The "$y_v$ region", which corresponds to interesting solutions $\boldsymbol{y}$ with certain constraints violated. This region is further split into subregions:

    - First subregion: This region is for solutions with only one constraint violated. A solution which violates exactly one constraint is mapped to the $i$-th box of this region. Since there are $m$ constraints, this region is of size $m$.

    - The remaining $N_{TB}/2 - m$ regions are further divided evenly into $Q - 1$ subregions. According to violated constraints $j_1, j_2, \ldots, j_p$ $(2 \leq p \leq Q)$, solution with $p$ constraints violated will be mapped to the $p$-th subregion, and have $N_H$ boxes corresponding to it, one for each hash function.

    - For each hash function $h^i$, $i \in \{1, 2, \ldots, N_H\}$, a set of $m$ positive integer values are chosen uniformly at random. This is done only once at the very beginning of the algorithm. Let the $i$-th set of such values be $\Phi^i = \{\phi_1^i, \phi_2^i, \ldots, \phi_m^i\}$. The $i$-th hash function is computed according to the following formula:

$$h^i[trace(\boldsymbol{y})] = (\sum_{k=1}^{p} \phi_{j_k}^i + \prod_{k=1}^{p} \phi_{j_k}^i) \mod (\frac{N_{TB}/2 - m}{Q - 1}), \;\; i \in \{1, \ldots, N_H\}$$

where *mod* operation denotes keeping the reminder of $(\sum_{k=1}^{p} \phi_{j_k}^{i} + \prod_{k=1}^{p} \phi_{j_k}^{i})$ divided by $(\frac{N_{TB}/2-m}{Q-1})$. The trace is computed by a combination of the set $\phi^i$ of random values based on the violated constraints' indices $j_1, \dots, j_p$, the *mod* operation ensures that the resulting index is within its suitable range of the $p$-th subregion. Interested readers could refer to S. Bakhtiari and Pieprzyk, 1995 [6] for a comprehensive treatment of this family of hash functions.

2. The "$y_w$ region", which corresponds to interesting solutions $\boldsymbol{y}$ with no violated constraints, but certain loose constraints. Similarly, this region is further split into subregions:

   - First subregion: this region is for solutions with only one loose constraint. A solution with exactly one loose constraint is mapped to the $i$-th box of this region. Since there are $m$ constraints, this region is of size $m$.
   - The remaining $N_{TB}/2 - m$ regions are further divided evenly into $Q - 1$ subregions. For each solution with loose constraints $j_1, j_2, \dots, j_p$ ($2 \leq p \leq Q$), we choose several subsets from those constraints. Each subset has 1, 2 or $r$ loose constraints ($r \leq p$). The numbers of such subsets, hereafter referred to as $N_1, N_2$, and $N_r$, respectively, become parameters of the algorithm. The subsets are chosen in a deterministic way, which means, for any particular trace, the same subsets are always chosen. Given a subset of indices $j_1, j_2, \dots, j_r$, we compute the trace index using one of the hash functions defined above. Note that we consider multiple subsets of indices, which implies that a trace is mapped to several trace boxes.

As to the implementation of the solution list, although we would eventually examine all the solutions in the list, obviously, it is more promising to find a better feasible solution in the neighborhood of the solution with a better objective and less number of violated constraints. We assign a priority value to each solution and extract the solution with the highest priority value. Thus, the priority queue is implemented as heap. Considering the insertion and/or extraction of interesting solution from the solution list using the principle of First In First Out (FIFO), with $O(1)$ computational complexity, using a priority queue has an $O(\log N_{TB})$ complexity during insertion and/or extraction of an interesting solution, plus an additional $O(N_{TB})$ storage of the priority values. However, from our observation, despite the downside for using a priority queue, it actually decreases the running time since the algorithm spent less time on examining less promising directions.

## 2.4 Computational Experiments

We summarize the set of parameters that are free to choose in our algorithm and we also specify the corresponding values we use in our experiments.

- $Q$ - the parameter determining what comprises an interesting solution.
- $MEM$ - the allocated memory for solution list. There is a lower bound for $MEM$ since we have to make sure $N_{TB}/2 - m > 0$, which ensures that the size of subregions in the trace box will be greater than zero.
- $N_H$ - the number of hash functions.
- $N_1, N_2, N_r$ - the number of subsets of $1, 2$, or $r$ loose constraints.

In order to simplify the benchmark of the algorithm, we fix the value of parameters $N_1, N_2, N_r$, and $N_H$: $N_1 = 2, N_2 = 2, N_r = 5$; $N_H = 2$. As to parameters $Q$ and $MEM$, we define a running sequence with steadily increasing values of $Q$ and $MEM$. Since the algorithm takes a general initial solution, at the beginning of the searching process, the solution is updated very frequently. If we cold start the algorithm with a large $Q$ and $MEM$ at the beginning, it may spend an unnecessary large computational time on clearing the solution list. The idea here is to use a warm start sequence as follows.

The following running sequence is referred to as running sequence I and it is used to test all the binary optimization instances in this chapter except other sequence is specified.

1. $Q = 4, MEM = 10$MB

2. $Q = 4, MEM = 50$MB

3. $Q = 6, MEM = 100$MB

4. $Q = 6, MEM = 250$MB

5. $Q = 10, MEM = 1$GB

6. $Q = 10, MEM = 2$GB

7. $Q = 15, MEM = 6$GB; $Q = 20, MEM = 6$GB

We use $\boldsymbol{z} = \mathbf{0}$ as the initial soluion for Step 1. In each of the following step, we use the solution from the previous step as the initial solution. In Step 7, two runs are performed sequentially, which means they are both started with the same feasible initial solution given by the output from the run in Step 6. And the run that gives the better result is chosen for analysis. The total running time is the sum of running time spent in all seven steps and the final result is given by the best run in Step 7.

We generate random set covering instances with the following tunable parameters.

- $m$: the number of constraints
- $n$: the number of binary variables
- $\boldsymbol{c}$: the cost vector
- $w$: the number of non-zero entries in each column of matrix $\boldsymbol{A}$
- $U[l, u]$: random integer with its value between $l$ and $u$

We generate 10 examples for each specific parameter settings listed in Table 2.1. For example, Instance *1.2* refers to the second example of a type one instance. We test our implementation of the algorithm on these instances and compare the results with the output from CPLEX 11.2. All of the tests are run on the Operations Research Center computational machines. Instances 1.1-1.10, 2.1-2.10, 3.1-3.10 are run on a machine with a Intel(R) Xeon(TM) CPU (3.00GHz, 2MB Cache), 8GB of RAM, and Ubuntu Linux operation system. Instances 4.1-4.10, 5.1-5.10, 6.1-6.10 are run on a machine with Intel(R) Xeon(R) CPU E5440 (2.83GHz, 6MB Cache), 8GB of RAM, and Ubuntu Linux operation system.

Table 2.1: Characteristics of set covering instances for IP

| Name | $m$ | $n$ | $c$ | $w$ |
|---|---|---|---|---|
| 1.1-1.10 | 1000 | 2500 | $e$ | 3 |
| 2.1-2.10 | 1000 | 2500 | $e$ | 5 |
| 3.1-3.10 | 1000 | 2500 | $e$ | $U[3,7]$ |
| 4.1-4.10 | 1000 | 2500 | $U[400,500]$ | 3 |
| 5.1-5.10 | 1000 | 2500 | $U[400,500]$ | 5 |
| 6.1-6.10 | 1000 | 2500 | $U[400,500]$ | $U[3,7]$ |

The results from Table 2.2 to Table 2.7 compare the objective obtained from our binary optimization algorithm versus CPLEX 11.2 after 5-hour, 10-hour and 20-hour computational time. ALG denotes our binary optimization algorithm, and it is run by using the running sequence I; CPX denotes CPLEX 11.2, and it is run with its default settings. In the remainder of the thesis, we emphasize the better results in bold font in the comparison Tables.

Table 2.2: Computational results on set covering instances 1.1-1.10

| | 5 hours | | 10 hours | | 20 hours | |
|---|---|---|---|---|---|---|
| Instance | ALG | CPX | ALG | CPX | ALG | CPX |
| 1.1 | 346 | **344** | 344 | 344 | 344 | **343** |
| 1.2 | 346 | **343** | 344 | **343** | 344 | **342** |
| 1.3 | 346 | **345** | **344** | 345 | 344 | 344 |
| 1.4 | **343** | 345 | **343** | 345 | **343** | 344 |
| 1.5 | **344** | 346 | 343 | 343 | 343 | **342** |
| 1.6 | **344** | 345 | **344** | 345 | 344 | **343** |
| 1.7 | 343 | 343 | 343 | 343 | 343 | **342** |
| 1.8 | **344** | 345 | **342** | 345 | **342** | 343 |
| 1.9 | **343** | 345 | **343** | 345 | **343** | 344 |
| 1.10 | **344** | 346 | **342** | 346 | **342** | 344 |

For instances 1.1-1.10 in Table 2.2, after 5 hours, there are 60% instances that our algorithm outperforms CPLEX, while 30% instances CPLEX outperforms. After 10 hours, there are 60% instances that our algorithm outperforms CPLEX, while 10% instances CPLEX outperforms. After 20 hours, there are 40% instances that our algorithm outperforms CPLEX and 50% instances CPLEX outperforms.

Table 2.3: Computational results on set covering instances 2.1-2.10

| | 5 hours | | 10 hours | | 20 hours | |
|---|---|---|---|---|---|---|
| Instance | ALG | CPX | ALG | CPX | ALG | CPX |
| 2.1 | **233** | 242 | **231** | 241 | **229** | 236 |
| 2.2 | **231** | 243 | **229** | 243 | **229** | 236 |
| 2.3 | **233** | 238 | **233** | 238 | **233** | 235 |
| 2.4 | **230** | 244 | **230** | 244 | **230** | 240 |
| 2.5 | **230** | 240 | **230** | 240 | **230** | 233 |
| 2.6 | **231** | 240 | **229** | 240 | **226** | 237 |
| 2.7 | **228** | 240 | **228** | 240 | **228** | 236 |
| 2.8 | **232** | 239 | **228** | 239 | **228** | 236 |
| 2.9 | **231** | 240 | **229** | 240 | **229** | 235 |
| 2.10 | **232** | 239 | **231** | 239 | **231** | 236 |

For instances 2.1-2.10 in Table 2.3, our algorithm outperforms CPLEX all the time.

Table 2.4: Computational results on set covering instances 3.1-3.10

| | 5 hours | | 10 hours | | 20 hours | |
|---|---|---|---|---|---|---|
| Instance | ALG | CPX | ALG | CPX | ALG | CPX |
| 3.1 | **226** | 231 | **226** | 231 | **226** | 226 |
| 3.2 | **228** | 231 | **226** | 231 | **226** | 229 |
| 3.3 | **228** | 235 | **227** | 235 | **227** | 231 |
| 3.4 | **228** | 231 | **225** | 231 | **225** | 229 |
| 3.5 | **231** | 235 | **229** | 235 | **227** | 230 |
| 3.6 | **230** | 234 | **227** | 234 | **227** | 229 |
| 3.7 | **228** | 236 | **225** | 236 | **224** | 228 |
| 3.8 | **226** | 234 | **225** | 234 | **225** | 230 |
| 3.9 | **231** | 234 | **225** | 233 | **222** | 229 |
| 3.10 | **231** | 232 | **225** | 232 | **222** | 227 |

For instances 3.1-3.10 in Table 2.4, our algorithm outperforms CPLEX all the time.

For instances 4.1-4.10 in Table 2.5, after 5 hours, there are 90% instances that our algorithm outperforms CPLEX, while 10% instances CPLEX outperforms. After 10 hours, our algorithm outperforms CPLEX for all instances. After 20 hours, there are 50% instances that our algorithm outperforms CPLEX and 50% instances CPLEX outperforms.

Table 2.5: Computational results on set covering instances 4.1-4.10

| | 5 hours | | 10 hours | | 20 hours | |
|---|---|---|---|---|---|---|
| Instance | ALG | CPX | ALG | CPX | ALG | CPX |
| 4.1 | **150453** | 151662 | **150085** | 150691 | 149615 | **149390** |
| 4.2 | **150557** | 150785 | **150110** | 150785 | **149230** | 149833 |
| 4.3 | **150782** | 151130 | **150233** | 151130 | 149923 | **149287** |
| 4.4 | 151789 | **150917** | **150062** | 150917 | **149486** | 150264 |
| 4.5 | **150449** | 151781 | **150404** | 151709 | 150233 | **149756** |
| 4.6 | **149449** | 149728 | **149449** | 149728 | 149449 | **148999** |
| 4.7 | **150337** | 151202 | **150337** | 151202 | 150337 | **148635** |
| 4.8 | **150088** | 151306 | **149860** | 150740 | **149503** | 149559 |
| 4.9 | **149676** | 150868 | **149609** | 150868 | **149293** | 149403 |
| 4.10 | **149791** | 150524 | **149608** | 150440 | **148703** | 149052 |

Table 2.6: Computational results on set covering instances 5.1-5.10

| | 5 hours | | 10 hours | | 20 hours | |
|---|---|---|---|---|---|---|
| Instance | ALG | CPX | ALG | CPX | ALG | CPX |
| 5.1 | **100264** | 107920 | **99663** | 107920 | **99663** | 103111 |
| 5.2 | **102454** | 107776 | **100943** | 107776 | **100131** | 102393 |
| 5.3 | **100266** | 107190 | **99837** | 105185 | **99837** | 100904 |
| 5.4 | **100393** | 106231 | **100393** | 106231 | **100393** | 101017 |
| 5.5 | **100341** | 107072 | **100180** | 107072 | **99911** | 102651 |
| 5.6 | **101272** | 106268 | **100585** | 106268 | **98989** | 101442 |
| 5.7 | **100718** | 107542 | **99978** | 107542 | **99978** | 102396 |
| 5.8 | **101070** | 108647 | **100530** | 108647 | **99651** | 103266 |
| 5.9 | **100592** | 106986 | **100288** | 106986 | **99970** | 103100 |
| 5.10 | **100084** | 108170 | **100084** | 108170 | **100084** | 102330 |

For instances 5.1-5.10 in Table 2.6, our algorithm outperforms CPLEX all the time.

Table 2.7: Computational results on set covering instances 6.1-6.10

| | 5 hours | | 10 hours | | 20 hours | |
|---|---|---|---|---|---|---|
| Instance | ALG | CPX | ALG | CPX | ALG | CPX |
| 6.1 | **99021** | 102026 | **98803** | 102026 | **98803** | 100951 |
| 6.2 | **98330** | 104147 | **98235** | 104147 | **98235** | 100533 |
| 6.3 | **99630** | 101245 | **99491** | 100789 | **98429** | 98552 |
| 6.4 | **99610** | 102623 | **98765** | 102623 | **97928** | 100997 |
| 6.5 | **99930** | 101656 | **99605** | 101404 | **98801** | 99790 |
| 6.6 | **99485** | 102107 | **98665** | 102104 | **98158** | 99624 |
| 6.7 | **99219** | 102449 | **99056** | 100877 | **98570** | 99128 |
| 6.8 | **99109** | 103281 | **98946** | 103281 | **98905** | 100709 |
| 6.9 | **100868** | 102188 | **99973** | 102188 | **99533** | 100930 |
| 6.10 | **100998** | 102991 | **100285** | 102991 | **100285** | 100837 |

For instances 6.1-6.10 in Table 2.7, our algorithm outperforms CPLEX all the time.

To conclude, for the set covering problems with 5 ones or 3 to 7 ones in each column of the constraint matrix $\boldsymbol{A}$, our binary optimization algorithm strictly outperforms CPLEX at all time points when both methods are run with the same amount of memory (6GB). While for set covering problems with 3 ones in each column of the constraint matrix $\boldsymbol{A}$, our algorithm is competitive with CPLEX. Therefore, our conclusion is that our algorithm outperforms CPLEX for denser binary optimization problems.

The selection of $Q$ and $MEM$ provides lots of flexibility for running the algorithm, and sequence I is not the best for every instance. We test another running sequence on instances 1.1-1.5, 2.1-2.5, 3.1-3.5, and refer to it as running sequence II.

1. $Q = 6, MEM = 1$GB

2. $Q = 10, MEM = 6$GB

The results in Table 2.8 show the advantage of using running sequence II. For 9 out of 15 instances, running sequence II converges faster and finds a better solution (smaller objective) for minimization problems. Meanwhile, both sequences gave the same quality solution for 4 out of 15 instances.

Table 2.8: Computational results using running sequence II

| | objective | | Computational Time(s) | |
|---|---|---|---|---|
| Instance | sequence II | sequence I | sequence II | sequence I |
| 1.1 | **342** | 344 | **22940.8** | 39381.7 |
| 1.2 | 344 | 344 | **23043.0** | 36469.2 |
| 1.3 | 344 | 344 | **25994.4** | 35399.3 |
| 1.4 | **342** | 343 | **27610.5** | 64542.9 |
| 1.5 | 343 | 343 | 35435.7 | **34907.1** |
| 2.1 | **228** | 229 | **36815.2** | 61555.4 |
| 2.2 | 230 | **229** | **25011.5** | 38057.0 |
| 2.3 | 234 | **233** | 28229.8 | **24039.8** |
| 2.4 | **228** | 230 | **51654.9** | 75694.7 |
| 2.5 | **229** | 230 | 45507.3 | **28857.4** |
| 3.1 | 226 | 226 | 41347.3 | **26754.3** |
| 3.2 | **224** | 226 | **31916.2** | 36932.4 |
| 3.3 | **226** | 227 | 36965.7 | **33791.1** |
| 3.4 | **224** | 225 | **40049.1** | 73840.2 |
| 3.5 | **226** | 227 | **31089.7** | 58954.4 |

We also test our algorithm on random set packing instances with the following tunable parameters.

- $m$: the number of constraints
- $n$: the number of binary variables
- $\boldsymbol{c}$: the cost vector
- $w$: the number of non-zeros in each column of matrix $\boldsymbol{A}$
- $U[l,u]$: random integer with its value between $l$ and $u$

We generate 5 examples for each specific parameter settings listed in Table 2.9. For example, Instance *1.2* refers to the second example of type one instance. We test our implementation of the algorithm on those instances and compare the results with the output from CPLEX 11.2. All the tests are run on the Operations Research

Center computational machine. Instances 1.1-1.10, 2.1-2.10, 3.1-3.10, 4.1-4.10 are run on a machine with a Intel(R) Xeon(TM) CPU (3.00GHz, 2MB Cache), 8GB of RAM, and Ubuntu Linux operation system. Instances 5.1-5.10, 6.1-6.10, 7.1-7.10, 8.1-8.10 are run on a machine with Intel(R) Xeon(R) CPU E5440 (2.83GHz, 6MB Cache), 8GB of RAM, and Ubuntu Linux operation system.

Table 2.9: Characteristics of set packing instances for IP

| Name | $m$ | $n$ | $c$ | $w$ |
|---|---|---|---|---|
| 1.1-1.5 | 1000 | 2500 | $e$ | 3 |
| 2.1-2.5 | 1000 | 2500 | $e$ | 5 |
| 3.1-3.5 | 1000 | 2500 | $e$ | 7 |
| 4.1-4.5 | 1000 | 2500 | $e$ | $U[3,7]$ |
| 5.1-5.5 | 1000 | 2500 | $U[400,500]$ | 3 |
| 6.1-6.5 | 1000 | 2500 | $U[400,500]$ | 5 |
| 7.1-7.5 | 1000 | 2500 | $U[400,500]$ | 7 |
| 8.1-8.5 | 1000 | 2500 | $U[400,500]$ | $U[3,7]$ |

For those examples, we consider the CPLEX parameter *MIP emphasis*, which controls the trade-offs between speed, feasibility, optimality, and moving bounds in solving MIP. With the default setting of *BALANCED [0]*, CPLEX works toward a rapid proof of an optimal solution, but balances that with effort toward finding high quality feasible solutions early in the optimization. When this parameter is set to *FEASIBILITY [1]*, CPLEX frequently will generate more feasible solutions as it optimizes the problem, at some sacrifice in the speed to the proof of optimality. When the parameter is set to *HIDDENFEAS [4]*, the MIP optimizer works hard to find high quality feasible solutions that are otherwise very difficult to find, so consider this setting when the *FEASIBILITY* setting has difficulty finding solutions of acceptable quality.

Since our algorithm also emphasizes on finding high quality feasible solution instead of proving optimality, we compare CPLEX's performance when the parameter *MIP emphasis* is set as 0, 1, and 4, in order to find out which setting will deliver the best solution. Based on the results in Table 2.10, we see that the results are consistently better than the default setting when *MIP emphasis* is set to be 4. Thus, in the following test, we run parallel experiments on CPLEX by setting the parameter *MIP* *emphasis* as 0 and 4, respectively, in order to make a better comparison between our algorithm and CPLEX.

Table 2.10: CPLEX performance with different *MIP emphasis* settings

| | 5 hours | | | 10 hours | | | 20 hours | | |
|---|---|---|---|---|---|---|---|---|---|
| Instance | 0 | 1 | 4 | 0 | 1 | 4 | 0 | 1 | 4 |
| 1.1 | 319 | **320** | 319 | 319 | **321** | **321** | 322 | **323** | 322 |
| 2.1 | 155 | 159 | **161** | 155 | 160 | **162** | 155 | **162** | **162** |
| 4.1 | 255 | 252 | **258** | 255 | 253 | **258** | 258 | 256 | **259** |
| 5.1 | 147663 | 144054 | **148706** | 147663 | 144234 | **148894** | 148617 | 146525 | **149500** |
| 6.1 | 71799 | 73722 | **74865** | 71799 | 73722 | **75488** | 75594 | 75407 | **76806** |
| 8.1 | 113760 | 110979 | **115852** | 113760 | 111668 | **115852** | 114948 | 114157 | **115909** |

In the following test results from Table 2.11 to Table 2.18, ALG denotes our binary optimization algorithm, and it is run by using running sequence I; CPX denotes CPLEX 11.2, and it is run with default settings; CPX(4) denotes CPLEX 11.2 run with *MIP emphasis* 4. We run CPX(4) on selective instances, thus the star mark in the tables denotes the result is not available because no experiment is performed. We compare the results from our algorithm with the best practice from CPLEX if the results from CPX(4) are available.

Table 2.11: Computational results on set packing instances 1.1-1.5

| | 5 hours | | | 10 hours | | | 20 hours | | |
|---|---|---|---|---|---|---|---|---|---|
| Instance | ALG | CPX | CPX(4) | ALG | CPX | CPX(4) | ALG | CPX | CPX(4) |
| 1.1 | **322** | 319 | 319 | **322** | 319 | 321 | **322** | **322** | **322** |
| 1.2 | 314 | 319 | **320** | 314 | 319 | **322** | 321 | 322 | **323** |
| 1.3 | 320 | 321 | **322** | 320 | 321 | **322** | 320 | **322** | **322** |
| 1.4 | **320** | 318 | 319 | 320 | 320 | **321** | 320 | **321** | **321** |
| 1.5 | **322** | 319 | 320 | **323** | 320 | 321 | **323** | **323** | 322 |

For instances 1.1-1.5 in Table 2.11, after 5 hours, there are 60% instances that our algorithm outperforms CPLEX, while 40% instances CPLEX outperforms. After 10 hours, there are 40% instances that our algorithm outperforms CPLEX, while 60% instances CPLEX outperforms. After 20 hours, there are 60% instances that CPLEX outperforms our algorithm and 40% ties.

For instances 2.1-2.5 in Table 2.12, our algorithm outperforms CPLEX all the time.

Table 2.12: Computational results on set packing instances 2.1-2.5

| | 5 hours | | | 10 hours | | | 20 hours | | |
|---|---|---|---|---|---|---|---|---|---|
| Instance | ALG | CPX | CPX(4) | ALG | CPX | CPX(4) | ALG | CPX | CPX(4) |
| 2.1 | **166** | 155 | 161 | **167** | 155 | 162 | **167** | 155 | 162 |
| 2.2 | **165** | 159 | 159 | **168** | 159 | 160 | **168** | 161 | 162 |
| 2.3 | **165** | 160 | 159 | **166** | 160 | 159 | **166** | 160 | 161 |
| 2.4 | **167** | 157 | 159 | **167** | 157 | 160 | **167** | 157 | 163 |
| 2.5 | **164** | 156 | 158 | **166** | 156 | 159 | **166** | 162 | 163 |

Table 2.13: Computational results on set packing instances 3.1-3.5

| | 5 hours | | | 10 hours | | | 20 hours | | |
|---|---|---|---|---|---|---|---|---|---|
| Instance | ALG | CPX | CPX(4) | ALG | CPX | CPX(4) | ALG | CPX | CPX(4) |
| 3.1 | **104** | 95 | 95 | **105** | 95 | 97 | **105** | 98 | 100 |
| 3.2 | **103** | 95 | 95 | **106** | 95 | 95 | **106** | 95 | 98 |
| 3.3 | **105** | 95 | 95 | **105** | 95 | 97 | **105** | 95 | 100 |
| 3.4 | **105** | 97 | 96 | **105** | 97 | 96 | **105** | 97 | 99 |
| 3.5 | **105** | 97 | 96 | **105** | 97 | 96 | **105** | 97 | 97 |

For instances 3.1-3.5 in Table 2.13, our algorithm outperforms CPLEX all the time.

Table 2.14: Computational results on set packing instances 4.1-4.5

| | 5 hours | | | 10 hours | | | 20 hours | | |
|---|---|---|---|---|---|---|---|---|---|
| Instance | ALG | CPX | CPX(4) | ALG | CPX | CPX(4) | ALG | CPX | CPX(4) |
| 4.1 | 255 | 255 | **258** | 255 | 255 | **258** | 255 | 258 | **259** |
| 4.2 | 253 | 255 | **260** | 255 | 255 | **261** | 255 | 257 | **261** |
| 4.3 | 251 | **256** | **256** | 253 | **256** | **256** | 253 | **258** | 256 |
| 4.4 | 249 | 252 | **255** | 249 | 252 | **255** | 249 | 253 | **256** |
| 4.5 | 256 | 258 | **259** | 256 | 258 | **259** | 256 | **259** | **259** |

For instances 4.1-4.5 in Table 2.14, CPLEX outperforms our algorithm all the time.

For instances 5.1-5.5 in Table 2.15, CPLEX outperforms for all the instances.

For instances 6.1-6.5 in Table 2.16, our algorithm outperforms CPLEX all the time.

For instances 7.1-7.5 in Table 2.17, our algorithm outperforms CPLEX all the time.

For instances 8.1-8.5 in Table 2.18, after 5 hours, CPLEX outperforms our algorithm all the time. After 10 hours and 20 hours, there are 80% instances that CPLEX

Table 2.15: Computational results on set packing instances 5.1-5.5

| | 5 hours | | | 10 hours | | | 20 hours | | |
|---|---|---|---|---|---|---|---|---|---|
| Instance | ALG | CPX | CPX(4) | ALG | CPX | CPX(4) | ALG | CPX | CPX(4) |
| 5.1 | 146896 | 147663 | **148706** | 147800 | 147663 | **148894** | 148905 | 148617 | **149500** |
| 5.2 | 147796 | 147914 | **149511** | 147796 | 147914 | **149726** | 147796 | 149116 | **149726** |
| 5.3 | 147951 | 147479 | **148125** | 147951 | 147479 | **148733** | 147951 | 148583 | **148788** |
| 5.4 | 147421 | 147023 | **148568** | 147421 | 147023 | **148889** | 147421 | 148399 | **148889** |
| 5.5 | 148545 | 148208 | **149104** | 148545 | 148869 | **149325** | 148545 | 149261 | **149823** |

Table 2.16: Computational results on set packing instances 6.1-6.5

| | 5 hours | | | 10 hours | | | 20 hours | | |
|---|---|---|---|---|---|---|---|---|---|
| Instance | ALG | CPX | CPX(4) | ALG | CPX | CPX(4) | ALG | CPX | CPX(4) |
| 6.1 | **75937** | 71799 | 74865 | **76590** | 71799 | 75488 | **77531** | 75594 | 76806 |
| 6.2 | **76928** | 72762 | 73080 | **77566** | 72762 | 73149 | **77566** | 74790 | 74752 |
| 6.3 | **76841** | 73447 | 74747 | **77681** | 73447 | 74747 | **77726** | 76086 | 75696 |
| 6.4 | **77475** | 72492 | 74392 | **77681** | 73231 | 74562 | **77681** | 75810 | 76299 |
| 6.5 | **76606** | 73361 | 73679 | **76985** | 73361 | 73750 | **77674** | 75376 | 74995 |

Table 2.17: Computational results on set packing instances 7.1-7.5

| | 5 hours | | | 10 hours | | | 20 hours | | |
|---|---|---|---|---|---|---|---|---|---|
| Instance | ALG | CPX | CPX(4) | ALG | CPX | CPX(4) | ALG | CPX | CPX(4) |
| 7.1 | **48200** | 43708 | 45204 | **48200** | 43708 | 45577 | **48200** | 46052 | 46187 |
| 7.2 | **48559** | 43548 | 44001 | **48895** | 44579 | 45809 | **48895** | 45486 | 47083 |
| 7.3 | **48019** | 43433 | 44602 | **48019** | 44658 | 45177 | **48019** | 46287 | 45606 |
| 7.4 | **48297** | 43667 | 44517 | **48297** | 43667 | 45272 | **48297** | 45721 | 47383 |
| 7.5 | **48721** | 44046 | 45280 | **48721** | 44046 | 46767 | **48721** | 44641 | 47545 |

Table 2.18: Computational results on set packing instances 8.1-8.5

| | 5 hours | | | 10 hours | | | 20 hours | | |
|---|---|---|---|---|---|---|---|---|---|
| Instance | ALG | CPX | CPX(4) | ALG | CPX | CPX(4) | ALG | CPX | CPX(4) |
| 8.1 | 114934 | 113760 | **115852** | 114934 | 113760 | **115852** | 114934 | 114948 | **115909** |
| 8.2 | 118242 | 117505 | **118261** | **118587** | 117505 | 118344 | **118587** | 118477 | 118447 |
| 8.3 | 116733 | 116896 | **118334** | 117115 | 116896 | **118334** | 117115 | 117686 | **118334** |
| 8.4 | 117227 | 116139 | **117317** | 117227 | 116421 | **117317** | 117227 | **117616** | 117317 |
| 8.5 | 114864 | 115134 | **116356** | 114864 | 115134 | **116356** | 114864 | 115915 | **116356** |

outperforms our algorithm and 20% instances our algorithm outperforms.

To conclude, for set packing problems with 5 ones in each column of the constraint matrix $\boldsymbol{A}$, our binary optimization algorithm strictly outperforms CPLEX at all time points when both methods are run with the same amount of memory (6GB). While for set packing problems with 3 ones or 3 to 7 ones in each column of the constraint matrix $\boldsymbol{A}$, our algorithm is competitive with CPLEX. To support the conclusion that our algorithm outperforms CPLEX for denser binary optimization problems, we introduce another type of denser instances 4.1-4.5, 8.1-8.5, which have 7 ones in each column of the matrix $\boldsymbol{A}$. The results are consistent with our assessment, that is, our algorithm outperforms CPLEX when the problem is dense. The new CPLEX setting *mip emphasis* does not change the comparison results for most of the examples.

We also try to do some modifications to the algorithm. Our first attempt is to skip Step 7 $SL \leftarrow \emptyset$, that is, we do not clear the solution list after we find a better solution. The computational results on set covering instances 1.1-6.1 are shown in Table 2.19. The algorithm converges within one run with parameters: $Q = 10, MEM = 1$GB. Here, we do not use a running sequence.

Table 2.19: Clear solution list versus maintain solution list

| | objective | | Computational time (s) | |
|---|---|---|---|---|
| Instance | Clear $SL$ | Maintain $SL$ | Clear $SL$ | Maintain $SL$ |
| 1.1 | 345 | 345 | 12661 | **3667** |
| 2.1 | **232** | 233 | 17083 | **5084** |
| 3.1 | 228 | **224** | 20241 | **7316** |
| 4.1 | **149616** | 150525 | 23393 | **4872** |
| 5.1 | **100571** | 100893 | 26573 | **6636** |
| 6.1 | 100367 | **99723** | 20416 | **4683** |

The results in Table 2.19 show that when we maintain the solution list, the algorithm converges much faster, due to the fact that clearing solution list requires significant computational time. On the other hand, the solution quality is comparable to the original version.

Another idea is to search all the adjacent solutions of a solution $\boldsymbol{x}$ instead of leaving the neighborhood when a better feasible solution is found. If there is at least one better feasible solution in the neighborhood of $\boldsymbol{x}$, we continue to search the neighborhood and keep the best feasible solution. In the next iteration, we start searching the neighborhood of the updated best feasible solution. Otherwise, we extract a new solution $\boldsymbol{x}$ from the solution list.

The outline of the new algorithm is as follows.

1. $\boldsymbol{x} = \boldsymbol{z}_0$; $SL = \boldsymbol{x}$

2. while $(SL \neq \emptyset)$

3. get a new solution $\boldsymbol{x}$ from $SL$

4. $\boldsymbol{Y} \leftarrow \boldsymbol{z}$

5. for each $(\boldsymbol{y}\,\text{adjacent to}\,\boldsymbol{x})$

6. if $(\boldsymbol{A}^T\boldsymbol{y} \geq \boldsymbol{b})$ & $(\boldsymbol{c}^T\boldsymbol{y} \leq \boldsymbol{c}^T\boldsymbol{Y})$

7. $\boldsymbol{Y} \leftarrow \boldsymbol{y}$

8. else if ($\boldsymbol{y}$ is interesting)

9. $TB[h(trace(\boldsymbol{y}))] \leftarrow \boldsymbol{c}^T\boldsymbol{y}$

10. $SL \leftarrow SL \cup \boldsymbol{y}$

11. if $(\boldsymbol{Y} \neq \boldsymbol{z})$

12. $SL \leftarrow \emptyset$

13. $SL \leftarrow SL \cup \boldsymbol{Y}$

14. go to step 3

15. else

16. go to step 3

The computational results on set covering instances 1.1-6.1, 1,2-6.2 are shown in Table 2.20 compared to the results from the original algorithm. Both algorithms are run using the parameters $Q = 10, MEM = 1$GB instead of using a running sequence.

Table 2.20: Comparison of the modified algorithm with the original algorithm

| | objective | | Computational time(s) | |
|---|---|---|---|---|
| Instance | ALG | New ALG | ALG | New ALG |
| 1.1 | 345 | 345 | **12661** | 13267 |
| 1.2 | 347 | 347 | 13091 | **12736** |
| 2.1 | 232 | 232 | 17083 | 17083 |
| 2.2 | 233 | 233 | 19449 | **19192** |
| 3.1 | 228 | 228 | **20241** | 21057 |
| 3.2 | 229 | 229 | 19441 | **19086** |
| 4.1 | **149616** | 150630 | **23393** | 26269 |
| 4.2 | 150994 | **149995** | **21211** | 30128 |
| 5.1 | 100571 | **99904** | 26573 | **25254** |
| 5.2 | **100362** | 101448 | 40129 | **24870** |
| 6.1 | 100367 | **100058** | **20416** | 34353 |
| 6.2 | 100694 | **100063** | 28395 | **24019** |

We have the following findings based on the results in Table 2.20, which show that the modified algorithm has competitive performance with the original algorithm.

1. The modified algorithm finds a better solution on some instances, for example, Instance 4.2, 5.1, 6.1, 6.2, while it converges to an inferior solution on Instance 4.1 and 5.2.

2. The modified algorithm converges faster on some instances, for example, Instance 1.2, 2.2, 3.2, 5.1, 5.2, 6.2, while it converges slower on Instance 1.1, 3.1, 4.1, 4.2 and 6.1.

# Chapter 3

# An Adaptive Local Search Algorithm for Mixed Integer Optimization

## 3.1 Problem Definition

We consider the following problem to test our mixed integer optimization algorithm.

$$
\begin{aligned}
\max\ & \boldsymbol{c}^T\boldsymbol{x} + \boldsymbol{d}^T\boldsymbol{y} \\
\text{s.t. } & \boldsymbol{A}\boldsymbol{x} + \boldsymbol{B}\boldsymbol{y} \le \boldsymbol{b} \\
& \boldsymbol{x} \in \{0,1\}^{n_1},\ \boldsymbol{y} \ge \boldsymbol{0}
\end{aligned}
\tag{3.1}
$$

where $\boldsymbol{A} \in \{0,1\}^{m \times n_1}$, $\boldsymbol{B} \in \{0,1\}^{m \times n_2}$, $\boldsymbol{b} \in \mathbb{Q}_+^m$, $\boldsymbol{c} \in \mathbb{Q}_+^{n_1}$, $\boldsymbol{d} \in \mathbb{Q}_+^{n_2}$.

This problem is referred to as the set packing mixed integer problems (MIP).

## 3.2 Algorithm

We test the binary optimization algorithm proposed by Bertsimas and Goyal [8]. The algorithm takes as inputs the matrices $\boldsymbol{A}$, $\boldsymbol{B}$, the vectors $\boldsymbol{b}$, $\boldsymbol{c}$, $\boldsymbol{d}$, parameters $Q$ and $MEM$, and an initial solution $(\boldsymbol{z}_{x_0}, \boldsymbol{z}_{y_0})$. The algorithm generates a series of feasible solutions with monotonically increasing objective values during the search process. The parameter $Q$ controls the search depth of the searching neighborhood, which poses a trade-off between solution quality and computational complexity. The parameter $MEM$ controls the size of the solution list, which affects the degree of collision of interesting solutions with similar characteristics in constraint violation and looseness.

We divide the solution into two parts: $\boldsymbol{x}$ and $\boldsymbol{y}(\boldsymbol{x})$, which stand for the solution to the binary variables and the solution to the continuous variables, respectively. As introduced in Section 2.2, we consider the following definition.

- $V(\boldsymbol{x}) = \max(\boldsymbol{A}\boldsymbol{x} - \boldsymbol{b}, \boldsymbol{0}) \in \mathbb{Z}_+^m$: the amount of constraint violation produced by $\boldsymbol{x}$.

- $U(\boldsymbol{x}) = \max(\boldsymbol{b} - \boldsymbol{A}\boldsymbol{x}, \boldsymbol{0}) \in \mathbb{Z}_+^m$: the amount of constraint looseness produced by $\boldsymbol{x}$.

- $W(\boldsymbol{x}) = \min(U(\boldsymbol{x}), \boldsymbol{e}) \in \{0,1\}^m$

- $trace(\boldsymbol{x}) = [V(\boldsymbol{x}); W(\boldsymbol{x}) - W(\boldsymbol{z}_x)] \in \mathbb{Z}_+^m \times \{0,1\}^m$, Let $\boldsymbol{z}_x$ to be the solution to the binary variables of the current best solution at a certain iteration of the algorithm.

- Two solutions $\boldsymbol{x}_1$ and $\boldsymbol{x}_2$ are said to be *adjacent* if $\boldsymbol{e}^T|\boldsymbol{x}_1 - \boldsymbol{x}_2| = 1$.

- A feasible solution $(\boldsymbol{z}_1, \boldsymbol{y}(\boldsymbol{z}_1))$ is said to be *better* than another feasible solution $(\boldsymbol{z}_2, \boldsymbol{y}(\boldsymbol{z}_2))$ if $(\boldsymbol{c}^T\boldsymbol{z}_1 + \boldsymbol{d}^T\boldsymbol{y}(\boldsymbol{z}_1)) - (\boldsymbol{c}^T\boldsymbol{z}_2 + \boldsymbol{d}^T\boldsymbol{y}(\boldsymbol{z}_2)) \geq 0.1$.

  The solution to the continuous variables is computed given the solution to the binary variables by solving

  $$\begin{cases} \arg\max\{\boldsymbol{d}^T\boldsymbol{y} \mid \boldsymbol{B}\boldsymbol{y} \leq \boldsymbol{b} - \boldsymbol{A}\boldsymbol{x}, \boldsymbol{y} \geq \boldsymbol{0}\}, & \boldsymbol{A}\boldsymbol{x} \leq \boldsymbol{b} \\ 0 & \boldsymbol{A}\boldsymbol{x} > \boldsymbol{b} \end{cases}$$

- A solution $\boldsymbol{x}$ is said to be *interesting* if the following three criteria hold:

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

  **(A2)** The number of violated constraints incurred by $\boldsymbol{x}$ is at most $Q$.

**(A3)** $(\boldsymbol{c}^T\boldsymbol{x} + \boldsymbol{d}^T\boldsymbol{y}(\boldsymbol{x})) - (\boldsymbol{c}^T\boldsymbol{x}' + \boldsymbol{d}^T\boldsymbol{y}(\boldsymbol{x}')) \geq 0.1$, $\forall \boldsymbol{x}'$already examined such that $h(trace(\boldsymbol{x})) = h(trace(\boldsymbol{x}'))$. $h(\cdot)$ is a linear function that maps a vector into an integer. The mapping is multiple-to-one, that is, different vectors may be mapped to the same integer. The specifications of $h(\cdot)$ are the same as the binary optimization algorithm in Section 2.3.

- A solution list $SL$

  The solution list stores the solutions to the binary variables, which satisfies the criteria of interesting solutions. They are ordered according to their assigned priority values. The definition of priority value is the same as the definition in Section 2.3. The solution with the highest priority is extracted first.

- A trace box $TB$

  The trace box entry $TB[i]$ stores the best objective among all of the interesting solutions $(\boldsymbol{x}, \boldsymbol{y}(\boldsymbol{x}))$ satisfying $h(trace(\boldsymbol{x})) = i$.

In our implementation, we change some specifications of the algorithm compared to the original one in paper [8].

We only consider violation and looseness incurred by the solution to the binary variables (equivalent to assuming all the continuous variables are zero). Thus, the definitions of $V(\boldsymbol{x}), W(\boldsymbol{x}), trace(\boldsymbol{x})$ are similar to the binary optimization algorithm. We use the criterion that the number of violated constraints instead of the total amount of violation cannot exceed $Q$. We update the best feasible solution and the interesting solution only when there is an numerical improvement of objective that is greater than or equal to 0.1. Here, we do not consider an improvement less than 0.1 appealing.

More specifically, we give an outline of the algorithm as follows.

Input: matrices $\boldsymbol{A}, \boldsymbol{B}$; vectors $\boldsymbol{b}, \boldsymbol{c}, \boldsymbol{d}$; feasible solution $(\boldsymbol{z}_{x_0}, \boldsymbol{z}_{y_0})$; scalar parameters $Q, MEM$

Output: best feasible solution $(\boldsymbol{z}_x, \boldsymbol{z}_y)$

1. $(\boldsymbol{z}_x, \boldsymbol{z}_y) = (\boldsymbol{z}_{x0}, \boldsymbol{z}_{y0})$; $SL = \boldsymbol{z}_x$ [$MEM$ is specified to determine the size of the $SL$]

2. while $(SL \neq \emptyset)$

3. get a new solution $\hat{\boldsymbol{z}}_x$ from $SL$

4. for each ($\boldsymbol{x}$ adjacent to $\hat{\boldsymbol{z}}_x$)

5. if $(\boldsymbol{A}\boldsymbol{x} \leq \boldsymbol{b})$

6. compute $\boldsymbol{y}(\boldsymbol{x})$ by solving $\arg\max\{\boldsymbol{d}^T\boldsymbol{y} \mid \boldsymbol{B}\boldsymbol{y} \leq \boldsymbol{b} - \boldsymbol{A}\boldsymbol{x}, \boldsymbol{y} \geq \boldsymbol{0}\}$

7. if $(\boldsymbol{c}^T\boldsymbol{x} + \boldsymbol{d}^T\boldsymbol{y}(\boldsymbol{x})) - (\boldsymbol{c}^T\boldsymbol{z}_x + \boldsymbol{d}^T\boldsymbol{y}(\boldsymbol{z}_y)) \geq 0.1$

8. $(\boldsymbol{z}_x,\ \boldsymbol{z}_y) \leftarrow (\boldsymbol{x},\ \boldsymbol{y}(\boldsymbol{x}))$

9. $\hat{\boldsymbol{z}}_x \leftarrow \boldsymbol{x}$

10. go to step 4

11. else if ($\boldsymbol{x}$ is interesting [$Q$ is specified for condition **A2**])

12. $TB[h(trace(\boldsymbol{x}))] \leftarrow \boldsymbol{c}^T\boldsymbol{x} + \boldsymbol{d}^T\boldsymbol{y}(\boldsymbol{x})$

13. $SL \leftarrow SL \cup \boldsymbol{x}$

14. else if ($\boldsymbol{x}$ is interesting [$Q$ is specified for condition **A2**])

15. $TB[h(trace(\boldsymbol{x}))] \leftarrow \boldsymbol{c}^T\boldsymbol{x} + \boldsymbol{d}^T\boldsymbol{y}(\boldsymbol{x})$, here $\boldsymbol{y}(\boldsymbol{x})$ is a zero vector

16. $SL \leftarrow SL \cup \boldsymbol{x}$

17. return $\boldsymbol{z}_x$

We divide the problem into a pure binary optimization subproblem and a linear optimization subproblem. We first solve the pure binary optimization subproblem $\max\{\boldsymbol{c}^T\boldsymbol{x} \mid \boldsymbol{A}\boldsymbol{x} \leq \boldsymbol{b}, \boldsymbol{x} \in \{0,1\}^{n_1}\}$ by using the similar idea as the binary optimization algorithm in Section 2.2. The continuous solution is computed given the solution to the binary variables by solving $\max\{\boldsymbol{d}^T\boldsymbol{y} \mid \boldsymbol{B}\boldsymbol{y} \leq \boldsymbol{b} - \boldsymbol{A}\boldsymbol{x}, \boldsymbol{y} \geq \boldsymbol{0}\}$. The algorithm starts with an initial feasible solution. In each iteration, the algorithm selects a candidate solution $\hat{\boldsymbol{z}}_x$ from the solution list $SL$ and examines all of its adjacent solutions. If any adjacent solution is interesting (refer to the definition of interesting solutions in Section 3.2), we store it in the solution list and update the appropriate trace boxes. If we find a better feasible solution $\boldsymbol{z}_x$, we jump to solution $\boldsymbol{z}_x$. The previous procedure resumes by examining the adjacent solutions of $\boldsymbol{z}_x$. Based on the results in Table 2.19, we maintain the solution list when a better feasible solution is found since it yields comparable results, but converges much faster compared to clearing the solution list.

## 3.3 Implementation Details

In order to utilize the memory efficiently, We use the following data structure to store the problem and the solution.

- We store the matrices, $\boldsymbol{A}, \boldsymbol{B}$ and the vectors $V(\boldsymbol{x}), U(\boldsymbol{x}), W(\boldsymbol{x})$ as sparse matrices and sparse vectors respectively, i.e., only store the indices and values of nonzero entries.

- We store only the binary solution $\boldsymbol{x}$ in the solution list not the corresponding continuous solution $\boldsymbol{y}$, because we only need the partial solution $\boldsymbol{x}$ to construct its neighborhood.

- We store the partial solution vector $\boldsymbol{x}$ in binary representation, which decreases the storage commitment for a solution $\boldsymbol{x}$ from $n$ to $(\frac{n}{\text{sizeof}(int)} + 1)$ integers.

- We use the same specifications of hash functions and the same method to extract a new solution from the solution list as described in Section 2.3.

When we search the neighborhood of a solution $\hat{\boldsymbol{z}}_x$ extracted from the solution list, if its adjacent solution $\boldsymbol{x}$ is feasible, we call CPLEX to solve the linear optimization subproblem $\max\{\boldsymbol{d}^T\boldsymbol{y} \mid \boldsymbol{B}\boldsymbol{y} \leq \boldsymbol{b} - \boldsymbol{A}\boldsymbol{x}, \boldsymbol{y} \geq \boldsymbol{0}\}$. Instead of solving it from scratch each time, we warm start the linear optimization subproblem. Notice that the adjacent solution only differs from the solution $\hat{z}_x$ by one unit for a single entry. Therefore, there is only slight change in the RHS of the linear optimization subproblem. Thus, we load the basis from solving the linear optimization subproblem associated with $\hat{z}_x$ to solve the linear optimization subproblems of its adjacent solutions using the dual simplex method.

Instead of solving the linear optimization subproblem to optimality from the very beginning, our algorithm solves each linear optimization subproblem to an adaptive level of accuracy that increases during the course of the algorithm. Here, $\delta$ denotes the level of accuracy to which CPLEX solve the linear optimization subproblem. We use the following values of $\delta$ as a function of time $t$ in hours:

$$\delta = \begin{cases} 0.1, & 0 \leq t \leq 5 \\ 0.05, & 5 < t \leq 10 \\ 0.01, & 10 < t \end{cases}$$

In Table 3.1, computational time records the time the algorithm takes to get to the objective. All of the instances are given a limited running time of 20 hours. Some active instances converge much earlier than 20 hours, some do not converge even after 20 hours. The characteristics of the instances are specified in Table 3.5 and 3.6. We use algorithm $\mathbb{A}2$ explained in Section 3.3 to compute the initial solution and allocate a small memory $MEM = 3Q$ MB with the $Q$ sequence explained in Section 3.4.

From the comparison in Table 3.1, we observe that for all the instances except Instance 3.1.1, starting from sub-optimality leads to a better solution, which shows that spending large computational effort on solving the linear optimization subproblem to optimality at the beginning does not bring much improvement to the quality of the solution.

We consider two algorithms to compute the initial solution for the mixed integer optimization problem. First, we introduce a greedy algorithm proposed by Bertsimas and Goyal [8], denoted as $\mathbb{A}$, to solve the pure binary set packing problem 3.2.

Table 3.1: Optimality tolerance for solving linear optimization subproblems

| | Computational time (s) | | objective | |
|---|---|---|---|---|
| Instance | Solve to optimality | Solve to sub-optimality | Solve to optimality | Solve to sub-optimality |
| 2.2.1 | 12742.61 | **2827.95** | 148.75 | **150.25** |
| 2.4.1 | **6084.64** | 9264.32 | 250.30 | **250.55** |
| 3.1.1 | 1001.81 | **130.18** | **283** | 282 |
| 3.3.1 | **5769.62** | 7361.79 | 450.27 | **455.13** |
| 4.1.1 | 49601.88 | **8989.42** | 537 | **538** |
| 4.3.1 | **52999.59** | 63604.78 | 845.45 | **849.06** |

$$\begin{aligned} &\max \boldsymbol{c}^T\boldsymbol{x} \\ &\text{s.t. } \boldsymbol{A}\boldsymbol{x} \leq \boldsymbol{b} \\ &\quad \boldsymbol{x} \in \{0,1\}^{n_1} \end{aligned} \tag{3.2}$$

where $\boldsymbol{A} \in \{0,1\}^{m\times n_1}, \boldsymbol{b} \in \mathbb{Z}_+^m, \boldsymbol{c} \in \mathbb{Q}_+^{n_1}$. We denote problem 3.2 as $\prod(\boldsymbol{A},\boldsymbol{b},\boldsymbol{c})$.

The greedy algorithm starts with $\boldsymbol{x} = \mathbf{0}$ as the initial solution. First it sorts the cost vector in a descending order, and examines the variable with the highest $c_j$. If we can increase this variable by one unit without violating any constraint, we keep this change and move to the variable with the next highest cost. Continue this process till we get to the variable with the lowest cost. We then get an initial solution to the problem 3.2. The outline of algorithm $\mathbb{A}$ is as follows.

Input: $\boldsymbol{A} \in \{0,1\}^{m\times n_1}, \boldsymbol{b} \in \mathbb{Z}_+^m, \boldsymbol{c} \in \mathbb{Q}_+^{n_1}$

Initialize $\boldsymbol{x} \leftarrow \mathbf{0}$.

1. Sort vector $\boldsymbol{c}$ with indices $I \leftarrow \{1,2,\dots,n_1\}$ in descending order, get new $\boldsymbol{c}'$ with index $\tau(i), \tau\{i\} \leftarrow \{1,2,\dots,n_1\}$

2. $\tau(i) \leftarrow 0$

3. while $(\tau(i) < n_1)$

4. If $\boldsymbol{A}(\boldsymbol{x}+\boldsymbol{e}_i) \leq \boldsymbol{b}$

5. $\quad \boldsymbol{x} \leftarrow \boldsymbol{x} + \boldsymbol{e}_i,$

6. $\quad \tau(i) \leftarrow \tau(i) + 1$

Then, we use the following algorithm, denoted as $\mathbb{A}_1$, to compute an initial feasible solution to the mixed integer problem 3.1.

1. Let $\boldsymbol{x}_1 \leftarrow \mathbb{A}(\prod(\boldsymbol{A}, \boldsymbol{b}, \boldsymbol{c}))$.

2. Let $\boldsymbol{y}_1 \leftarrow \arg\max\{\boldsymbol{d}^T\boldsymbol{y} \,|\, \boldsymbol{B}\boldsymbol{y} \leq \boldsymbol{b} - \boldsymbol{A}\boldsymbol{x}_1, \boldsymbol{y} \geq \boldsymbol{0}\}$

3. Let $\boldsymbol{y}_2 \leftarrow \arg\max\{\boldsymbol{d}^T\boldsymbol{y} \,|\, \boldsymbol{B}\boldsymbol{y} \leq \boldsymbol{b}, \boldsymbol{y} \geq \boldsymbol{0}\}$.

4. Let $\boldsymbol{x}_2 \leftarrow \mathbb{A}\big(\prod(\boldsymbol{A}, (\boldsymbol{b} - \boldsymbol{B}\boldsymbol{y}_2), \boldsymbol{c})\big)$

5. If $(\boldsymbol{c}^T\boldsymbol{x}_1 + \boldsymbol{d}^T\boldsymbol{y}_1 \geq \boldsymbol{c}^T\boldsymbol{x}_2 + \boldsymbol{d}^T\boldsymbol{y}_2)$, return $(\boldsymbol{x}_1, \boldsymbol{y}_1)$; otherwise, return $(\boldsymbol{x}_2, \boldsymbol{y}_2)$

If the solution $(\boldsymbol{x}_1, \boldsymbol{y}_1)$ is chosen as the initial solution, we say that the binary variables dominate the initial solution. Usually, in the following search process, the objective contributed by binary variables will decrease and the objective contributed by continuous variables will increase, and vise versa for the case that the solution $(\boldsymbol{x}_2, \boldsymbol{y}_2)$ is chosen as the initial solution.

We also consider the following algorithm, denoted as $\mathbb{A}_2$, to compute the initial solution. The main difference is that we use the binary optimization algorithm in Section 2.2 to solve the binary optimization subproblem 3.2.

1. Let $\boldsymbol{x}_1 \leftarrow \mathbb{B}(\prod(\boldsymbol{A}, \boldsymbol{b}, \boldsymbol{c}))$, the parameters $Q$ and $MEM$ of algorithm $\mathbb{B}$ are set as 4 and 800 MB, respectively.

2. Let $\boldsymbol{y}_2 \leftarrow \arg\max\{\boldsymbol{d}^T\boldsymbol{y} \,|\, \boldsymbol{B}\boldsymbol{y} \leq \boldsymbol{b}, \boldsymbol{y} \geq \boldsymbol{0}\}$.

3. If $(\boldsymbol{c}^T\boldsymbol{x}_1 \geq \boldsymbol{d}^T\boldsymbol{y}_2)$, return $(\boldsymbol{x}_1, \boldsymbol{0})$; otherwise, return $(\boldsymbol{0}, \boldsymbol{y}_2)$

If the solution $(\boldsymbol{x}_1, \boldsymbol{0})$ is chosen as the initial solution, we also say that the binary variables dominate the initial solution, and vise versa.

We run several experiments to compare algorithm $\mathbb{A}_1$ and $\mathbb{A}_2$, and present the results in Table 3.3.

Table 3.2: Characteristics of algorithms to compute the initial solution

| | $\mathbb{A}_1$ | | $\mathbb{A}_2$ | |
|---|---|---|---|---|
| Instance | objective of the initial solution | objective after 20 hours | objective of the initial solution | objective after 20 hours |
| 2.2.1 | 135.97 | 141.36 | **141** | **148.75** |
| 2.4.1 | 218.66 | **255.95** | **237.34** | 250.55 |
| 3.1.1 | 218 | 281 | **281** | **282** |
| 3.3.1 | 394.93 | **457.01** | **449.19** | 455.13 |
| 4.1.1 | 422 | 511 | **531** | **538** |
| 4.3.1 | 727.54 | **810.57** | 834.76 | **849.06** |

Table 3.3: Algorithm comparison for computing the initial solution

| | $\mathbb{A}1$ | | | $\mathbb{A}2$ | | |
|---|---|---|---|---|---|---|
| Instance | 5 hours | 10 hours | 20 hours | 5 hours | 10 hours | 20 hours |
| 2.2.1 | 140.11 | 140.84 | 141.36 | **148.75** | **148.75** | **148.75** |
| 2.4.1 | **254.98** | **255.95** | **255.95** | 250.55 | 250.55 | 250.55 |
| 3.1.1 | 281 | 281 | 281 | **282** | **282** | **282** |
| 3.3.1 | 450.59 | **456.71** | **457.01** | **455.13** | 455.13 | 455.13 |
| 4.1.1 | 468 | 486 | 511 | **538** | **538** | **538** |
| 4.3.1 | 780.78 | 791.65 | 810.57 | **845.26** | **846.31** | **849.06** |

The characteristics of the instances we test here are specified in Table 3.5 and 3.6. We solve the linear optimization subproblem to an adaptive level of accuracy as explained in Section 3.3 and allocate a small memory $MEM = 3Q$ MB with the $Q$ sequence explained in Section 3.4. Table 3.3 shows more results about the objective after 5 hours and 10 hours besides the results after 20 hours presented in Table 3.2.

From Table 3.3, we observe that for Instance 2.2.1, 3.1.1, 4.1.1, 4.3.1, using algorithm $\mathbb{A}_2$ to compute an initial solution delivers much better final results, while for Instance 2.4.1, 3.3.1, algorithm $\mathbb{A}_2$ slightly underperforms $\mathbb{A}_1$. Note that for all instances except Instance 2.2.1, the binary variables dominate the initial solution computed by algorithm $\mathbb{A}_1$; and for all instances, the binary variables also dominate the initial solution computed by algorithm $\mathbb{A}_2$. In order to understand the advantage of algorithm $\mathbb{A}_2$, from the results in Table 3.2, we observe that algorithm $\mathbb{A}_2$ reaches a much better initial objective, which explains the following better performance, this is expected as greedy is naive. For Instance 2.4.1 and 3.3.1, the initial solution with lower objective computed by algorithm $\mathbb{A}_1$ might give more space for the continuous variable to take positive values, thus the final results after 20 hours are better.

To conclude, using algorithm $\mathbb{A}_2$ to compute the initial solution for the mixed integer optimization algorithm delivers a better final results on most of the instances we test.

## 3.4 Computational Experiments

We generate random set packing instances for MIP with the following tunable parameters.

- $m$: the number of constraints
- $n_1$: the number of integer variables
- $n_2$: the number of integer variables
- $\boldsymbol{c}_1$: the cost vector for integer variables
- $\boldsymbol{c}_2$: the cost vector for continuous variables
- $w_1$: the number of non-zeros in each column of matrix $\boldsymbol{A}$
- $w_2$: the number of non-zeros in each column of matrix $\boldsymbol{B}$
- $\boldsymbol{b}$: the RHS of the constraints, here it is set to $\boldsymbol{e}$
- $U[l, u]$: random number with its value between $l$ and $u$

We generat the following instances with parameter settings specified in Table 3.4, 3.5, 3.6, 3.7. Instance *1.2* denotes type I instance with the second type of parameter settings. Instance *1.2.2* denotes example two of type 1.2 instance.

Table 3.4: Characteristics of set packing instances for MIP, type I

| Name | $m$ | $n_1$ | $n_2$ | $\boldsymbol{c}_1$ | $\boldsymbol{c}_2$ | $w_1$ | $w_2$ |
|---|---|---|---|---|---|---|---|
| 1.1 | 1000 | 1250 | 1250 | $\boldsymbol{e}$ | $\boldsymbol{e}$ | 3 | 3 |
| 1.2 | 1000 | 1250 | 1250 | $\boldsymbol{e}$ | $\boldsymbol{e}$ | 5 | 5 |
| 1.3 | 1000 | 1250 | 1250 | $\boldsymbol{e}$ | $\boldsymbol{e}$ | 7 | 7 |

The first type of instances specified in Table 3.4 have 1000 constraints, 1250 binary variable, and 1250 continuous variables. The constraint matrix has the same density for both binary variable and continuous variables, i.e., $w_1 = w_2$. And the cost of all variables are one, i.e., $\boldsymbol{c}_1 = \boldsymbol{c}_2 = \boldsymbol{e}$.

Table 3.5: Characteristics of set packing instances for MIP, type II

| Name | $m$ | $n_1$ | $n_2$ | $c_1$ | $c_2$ | $w_1$ | $w_2$ |
|---|---|---|---|---|---|---|---|
| 2.1 | 1000 | 1250 | 1250 | $\boldsymbol{e}$ | $0.75\boldsymbol{e}$ | 3 | 3 |
| 2.2 | 1000 | 1250 | 1250 | $\boldsymbol{e}$ | $0.75\boldsymbol{e}$ | 5 | 5 |
| 2.3 | 1000 | 1250 | 1250 | $U[1,2]$ | $0.75 \cdot U[1,2]$ | 3 | 3 |
| 2.4 | 1000 | 1250 | 1250 | $U[1,2]$ | $0.75 \cdot U[1,2]$ | 5 | 5 |

The second type of instances specified in Table 3.5 have 1000 constraints, 1250 binary variable, and 1250 continuous variables. The constraint matrix has the same density for both binary variable and continuous variables, while the cost coefficients of continuous variables are smaller than binary variables, i.e., $\boldsymbol{c}_1 = \boldsymbol{e}, \boldsymbol{c}_2 = 0.75\boldsymbol{e}$.

Table 3.6: Characteristics of set packing instances for MIP, type III

| Name | $m$ | $n_1$ | $n_2$ | $c_1$ | $c_2$ | $w_1$ | $w_2$ |
|---|---|---|---|---|---|---|---|
| 3.1 | 1000 | 1250 | 1250 | $\boldsymbol{e}$ | $\boldsymbol{e}$ | 3 | 5 |
| 3.2 | 1000 | 1250 | 1250 | $\boldsymbol{e}$ | $\boldsymbol{e}$ | 5 | 9 |
| 3.3 | 1000 | 1250 | 1250 | $U[1,2]$ | $U[1,2]$ | 3 | 5 |
| 3.4 | 1000 | 1250 | 1250 | $U[1,2]$ | $U[1,2]$ | 5 | 9 |

The third type of instances specified in Table 3.6 have 1000 constraints, 1250 binary variable, and 1250 continuous variables. The cost of both variables are at the same scale, while the constraint matrix for binary variable is sparser than for continuous variables, i.e., $w_1 = 3, w_2 = 5$.

Table 3.7: Characteristics of set packing instances for MIP, type IV

| Name | $m$ | $n_1$ | $n_2$ | $c_1$ | $c_2$ | $w_1$ | $w_2$ |
|---|---|---|---|---|---|---|---|
| 4.1 | 2000 | 2000 | 2000 | $\boldsymbol{e}$ | $\boldsymbol{e}$ | 3 | 5 |
| 4.2 | 2000 | 2000 | 2000 | $\boldsymbol{e}$ | $\boldsymbol{e}$ | 5 | 9 |
| 4.3 | 2000 | 2000 | 2000 | $U[1,2]$ | $U[1,2]$ | 3 | 5 |
| 4.4 | 2000 | 2000 | 2000 | $U[1,2]$ | $U[1,2]$ | 5 | 9 |

The fourth type of instances specified in Table 3.7 have 2000 constraints, 2000 binary variable, and 2000 continuous variables. The cost of both variables are at the same scale, while the constraint matrix for binary variable is sparser than for continuous variables.

We summarize the set of parameters that are free to choose in our algorithm and also specify the corresponding values we use in our experiments.

- $Q$ - the parameter determining what comprises an interesting solution.
- $MEM$ - the allocated memory for solution list. There is a lower bound for $MEM$ since we have to make sure $N_{TB}/2 - m > 0$, which ensures that the size of subregions in the trace box will be greater than zero.
- $N_H$ - the number of hash functions.
- $N_1, N_2, N_r$ - the number of subsets of $1, 2$, or $r$ loose constraints.

The values of $N_1, N_2, N_r$, and $N_H$ are the same as the binary optimization algorithm: $N_1 = 2, N_2 = 2, N_r = 5; N_H = 2$. For parameters $Q$ and $MEM$, $Q$ controls the trade-off between the algorithm complexity and the solution quality. The larger the $Q$, the larger the chance to find a better solution; parameter $MEM$ determines how much memory we allocate for the solution list. We use an adaptive value for $Q$ by starting with a small value of $Q = 4$ and gradually increasing the value of $Q$ as the algorithm progresses. We use the following sequence of values for $Q$ as a function of the elapsed time t (in hours):

$$Q = 4 + 2i,\ 2i \leq t \leq 2i + 2,\ i = 0, 1, \ldots, 9$$

When $Q$ increases, it is better to increase the size of the solution list accordingly, so that more solutions will be stored as interesting solutions. To understand how the amount of memory allocated affects algorithmic performance, we perform experiments to compare the algorithmic performance with large memory allocation, $MEM = 200Q$ MB, versus small memory allocation, $MEM = 3Q$ MB.

In Table 3.8, computational time records the time the algorithm takes to get to the objective. All of the instances are given a limited running time of 20 hours. Some active instances converge much earlier than 20 hours, some do not converge even after 20 hours. We solve the linear optimization subproblem to an adaptive level of accuracy as explained in Section 3.3 and use algorithm $\mathbb{A}2$ explained in Section 3.3 to compute the initial solution.

The results in Table 3.8 show that all the instances except Instance 4.3.1 exhibit better performance with large memory. However, for Instance 4.3.1, we observe a better performance with small memory. As to the computational time, a large memory allocation takes a longer time to converge for Instance 2.2.1, 2.4.1, 3.1.1, 3.3.1. To conclude, assigning a large memory such as $MEM = 200Q$ MB, delivers a better overall performance, at the expense of longer running time.

Table 3.8: Large memory versus small memory

| | Computational time (s) | | objective | |
|---|---|---|---|---|
| Instance | large memory | small memory | large memory | small memory |
| 2.2.1 | 64443.16 | **2827.95** | **152.75** | 150.25 |
| 2.4.1 | 16780.35 | **9264.32** | **253.32** | 250.55 |
| 3.1.1 | 26750.90 | **130.18** | **285** | 282 |
| 3.3.1 | 27760.40 | **7361.79** | **457.12** | 455.13 |
| 4.1.1 | **2773.81** | 8989.42 | **539** | 538 |
| 4.3.1 | **54568.29** | 63604.78 | 847.11 | **849.06** |

We test our implementation of the mixed integer optimization algorithm on all types of instances we generate and compare the results with the output from CPLEX 11.2. All the tests are run on the Operations Research Center computational machines. The results from Table 3.9 to Table 3.12 compare the objective obtained from our mixed integer optimization algorithm versus CPLEX 11.2 after 5-hour, 10-hour and 20-hour computational time. MIP denotes our mixed integer optimization algorithm. The initial solution is solved by algorithm $\mathbb{A}2$. The memory allocation is large: $MEM = 200Q$ MB. CPX denotes CPLEX 11.2, and it runs with its default settings.

Table 3.9: Computational results on set packing instances for MIP, type I

| | 5 hours | | 10 hours | | 20 hours | | MIP | |
|---|---|---|---|---|---|---|---|---|
| Instance | MIP | CPX | MIP | CPX | MIP | CPX | cx | dy |
| 1.1.1 | 312.13 | **318** | 312.33 | **318** | 313.08 | **318** | 72 | 241.08 |
| 1.2.1 | **183.87** | 183.50 | **183.87** | 183.50 | 183.87 | **183.96** | 10 | 173.87 |
| 1.3.1 | 132.89 | 132.89 | 132.89 | 132.89 | 132.89 | 132.89 | 0 | 132.89 |

In Table 3.9, cx denotes objective contributed by binary variables and dy denotes objective contributed by continuous variables; cx and dy adds up to MIP. Based on the results in Table 3.9, we observe that the integer variables do not contribute much to the objective. In addition, these instances all start with an initial solution dominated by the continuous variables. We also examine the objective from solving the linear optimization subproblem $\max\{\boldsymbol{d}^T\boldsymbol{y} \mid \boldsymbol{B}\boldsymbol{y} \leq \boldsymbol{b} - \boldsymbol{A}\boldsymbol{x}_1, \boldsymbol{y} \geq \boldsymbol{0}\}$ initially, which are 297.37, 182.54, 132.89, respectively. Therefore, we conclude that the solution do not go further beyond solving the linear optimization subproblem at the beginning, especially for Instances 1.2.1 and 1.3.1. In order to better compare the capability of solving the binary part of the MIP, we decide to test the following types of instances: decrease the cost associated with continuous variables, where the results are shown in Table 3.10; increase the constraint matrix's density of continuous variables, where the results are shown in Table 3.11; increase both the constraint matrix's density of continuous variables and the problem size, where the results are shown in Table 3.12.

Table 3.10: Computational results on set packing instances for MIP, type II

| | 5 hours | | 10 hours | | 20 hours | |
|---|---|---|---|---|---|---|
| Instance | MIP | CPX | MIP | CPX | MIP | CPX |
| 2.1.1 | 292.5 | **301.5** | 292.5 | **301.5** | 292.5 | **303** |
| 2.1.2 | 291 | **299** | 291.75 | **299** | 291.75 | **301.25** |
| 2.2.1 | **154.25** | 152 | **155.5** | 152 | 156.25 | **157** |
| 2.2.2 | **153.50** | 149.88 | **154.75** | 149.88 | **155.25** | 154.25 |
| 2.3.1 | 468.44 | **487.46** | 469.70 | **487.46** | 469.70 | **489.67** |
| 2.3.2 | 466.68 | **482.59** | 467.71 | **482.70** | 468.76 | **484.03** |
| 2.4.1 | **253.32** | 250.26 | **253.32** | 250.26 | 253.32 | **257.56** |
| 2.4.2 | **251.03** | 246.42 | **251.18** | 246.42 | 251.18 | **255.47** |

Table 3.11: Computational results on set packing instances for MIP, type III

| | 5 hours | | 10 hours | | 20 hours | |
|---|---|---|---|---|---|---|
| Instance | MIP | CPX | MIP | CPX | MIP | CPX |
| 3.1.1 | 283 | **288** | 285 | **288** | 285 | **290** |
| 3.1.2 | 282 | **292** | 282 | **292** | 282 | **293** |
| 3.2.1 | **148** | 142 | **148** | 142 | **153** | 151 |
| 3.2.2 | **150.51** | 149.26 | **150.51** | 149.67 | **150.51** | 149.99 |
| 3.3.1 | 456.61 | **461.59** | 457.12 | **461.59** | 457.12 | **465.05** |
| 3.3.2 | 459.03 | **484.59** | 459.63 | **484.59** | 459.63 | **486.55** |
| 3.4.1 | **250.50** | 234.89 | **250.50** | 234.89 | **250.50** | 244.95 |
| 3.4.2 | **236.38** | 233.97 | **236.38** | 233.97 | 236.38 | **236.60** |

Table 3.12: Computational results on set packing instances for MIP, type IV

| | 5 hours | | 10 hours | | 20 hours | |
|---|---|---|---|---|---|---|
| Instance | MIP | CPX | MIP | CPX | MIP | CPX |
| 4.1.1 | 539 | **542** | 539 | **542** | 539 | **548** |
| 4.1.2 | 523.58 | **560** | 530.70 | **560** | 533.50 | **563** |
| 4.2.1 | **283** | 274 | **283** | 274 | 283 | **285** |
| 4.2.2 | **285.95** | 285.12 | **286.27** | 285.12 | **286.48** | 285.65 |
| 4.3.1 | 845.31 | **857.49** | 845.89 | **857.49** | 847.11 | **863.38** |
| 4.3.2 | 844.13 | **891.00** | 847.57 | **891.00** | 851.27 | **902.35** |
| 4.4.1 | **459.99** | 444.89 | **460.02** | 444.89 | **460.16** | 457.33 |
| 4.4.2 | **438.27** | 433.36 | **438.27** | 433.36 | **438.27** | 435.90 |

Based on the results shown in Table 3.10, 3.11, 3.12, our assessment is that when both methods are run with the same amount of memory (around 4GB), the mixed integer optimization algorithm outperforms CPLEX most of the time on set packing problems that have 5 ones in each column of the constraint matrix $\boldsymbol{A}$ for the binary variables, that is, type 2.2, 2.4, 3.2, 3.4, 4.2, 4.4 instances. While for the other sparser instances, our algorithm delivers results that are close to CPLEX. The largest gap is around 5% relative to the result from CPLEX. Thus, further work to enhance the performance of our mixed integer optimization algorithm on sparser problems is strongly desirable.

# Chapter 4

# Conclusions

In the thesis, we implement two algorithms for binary optimization and mixed integer optimization, respectively. We investigate the proper parameter settings to achieve satisfactory algorithmic performance. Furthermore, we test those two algorithms on large amount of randomly generated fairly large-size instances and compare their performance with the leading optimization package CPLEX 11.2.

For the binary optimization algorithm, our findings are as follows.

1. We introduce a warm start running sequence to achieve good algorithmic performance. We try two running sequences with gradually increasing values of parameters $Q$ and $MEM$. From the computational results, those sequences are able to get to a better or competitive solution compared to CPLEX.

2. We compare the performance of our algorithm with CPLEX on different types of instances. Our binary optimization algorithm strictly outperforms CPLEX on sparse instances, such as set covering problems with 5 ones or 3 to 7 ones in each column of the constraint matrix $\boldsymbol{A}$, at all time points when both methods are run with the same amount of memory (6GB). While on moderately dense instances, such as set packing problems with 3 ones or 3 to 7 ones in each column of the constraint matrix $\boldsymbol{A}$, our algorithm is competitive with CPLEX.

3. We also try to modify the algorithm in the following aspects: to maintain the solution list instead of clearing it after a better solution is found; to keep search-

ing the neighborhood instead of leaving it after a better solution is found. Our findings show that maintaining the solution list has shorten the computational time without jeopardizing solution quality. As to the latter modification, there is not much performance difference compared to the original algorithm.

For the mixed integer optimization algorithm, our findings are as follows.

1. We compare two algorithms to compute an initial solution. We find that algorithm $\mathbb{A}2$ delivers a better final solution most of the time.

2. Following the idea of warm start from the binary optimization algorithm, we gradually increase $Q$ with respect to the elapsed computational time during the course of the algorithm. We notice that assigning a large memory delivers better results most of the time.

3. We also investigate the difference between solving the linear subproblem to optimality and to an adaptive level of accuracy. We find that the algorithm performs better when we solve the linear subproblem to an adaptive level of accuracy.

4. We test our algorithm on different types of instances and compare the performance with CPLEX. Based on the experimental results, our assessment is that our mixed integer optimization algorithm outperforms CPLEX most of the time on moderately dense set packing instances, for example, problems with 5 ones in each column of the constraint matrix $\boldsymbol{A}$ for the integer variables. While for sparse instances, our algorithm delivers results that are close to CPLEX. The largest gap is around 5% relative to the result from CPLEX. Thus, further work to enhance the performance of our mixed integer optimization algorithm on sparser problems is strongly desirable.

# Bibliography


[1] E. Aarts, J. K. Lenstra, eds. 1997. *Local Search in Combinatorial Optimization.* John wiley & Sons, Inc., New York, NY, USA

[2] E. Balas, S. Ceria, G. Cornuéjols. 1993. A lift-and-project cutting plane algorithm for mixed 0-1 programs. *Math. Program.* **58** 295-324

[3] E. Balas, C.H. Martin. 1980. Pivot and complement-a heuristic for 0-1 programming. *Management Science* **26** 86-96.

[4] J. Eckstein, M. Nediak. 2007. Pivot, cut, and dive: a heuristic for 0-1 mixed integer programming. *Journal of Heuristics* **13** 471 503.

[5] M. Fischetti, F. Glover, A. Lodi. 2005. The feasibility pump. *Math. Program.* **104** 91-104.

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

[7] D. Bertsimas, D. Iancu, D. Katz. 2008. A General Purpose Local Search Algorithm for Binary Optimization. *Submitted for publication.*

[8] D. Bertsimas, V. Goyal. An Adaptive Local Search Algorithm for Solving Mixed Integer Optimization Problems. *Submitted for publication.*