LEHIGH UNIVERSITY

418 FINAL PROJECT

A review of mixed integer knapsack problems

Author:
Chenxin Ma, Xi He

Supervisor:
Prof. Ted Ralphs

Fall 2014

Contents

1   Introduction

Consider a positive integer n, letting $b \in \mathbb{Q}$, $a \in \mathbb{Q}^n$, $l \in \{\mathbb{Q} \cup \{-\infty\}\}^n$, $u \in \{\mathbb{Q} \cup \{+\infty\}\}^n$ and $I \subset [n] := \{1, ..., n\}$. The Mixed Integer Knapsack Set is defined as

$$K = \{x \in \mathbb{R}^n : a^T x \le b, l \le x \le u, x_i \in \mathbb{Z}, \forall i \in I\}. \tag{1}$$

Furthermore, if we have $c \in \mathbb{Q}^n$ and assume l_i is finite for each $i \in [n]$, then the Mixed Integer Knapsack Problem (MIKP) can be described as:

$$\max\{c^T x : x \in K\}. \tag{2}$$

We assume that for each $k = 1, 2, \ldots, n$ either $a_k \ne 0$ or $c_k \ne 0$, otherwise, we could remove variable x_k without affecting the problem.

In this report, we present a new branch-and-bound algorithm for MIKP. The methodology that we propose is a linear-programming-based algorithm which exploits dominance conditions. We further make use of lexicographic-domination conditions to eliminate problems with symmetry. One interesting aspect of this approach is that it differs from traditional linear-programming based algorithms by allowing feasible solutions to be pruned during the branching phase.

It might be very difficult to solve MIKP, even by using very effective mixed integer programming solvers such as CPLEX [1]. However, the proposed algorithm is shown to be very effective in solving instances of MIKP, much more effectice than CPLEX in fact, both in the amount of time taken to solve problems as by the size of the branch and bound tree explored to find the optimal solution.

In the following content, we will state procedures aiming to solve MIKP

1. An easy way of identifying unbounded solutions.

2. A way of pre-processing instances of MIKP.

3. The issue of quickly solving the LP-relaxation of MIKP.

4. A simple branch-and-bound algorithm for MIKP.

5. A enhancement of the branch-and-bound algorithm by introducing domination-criteria.

And finally, we will analyze the computational results of our algorithm and compare it with the general mixed-integer-programing solver CPLEX.

2   Algorithms for solving MIKP

2.1   Infeasible, unbounded, and trivial instances of MIKP

We aim to use a simple procedure to verify that our MIKP instances are either infeasible, unbounded or trivial, where 'trivial' stands for that our instances are very easy to solve.

Actually, it is very easy to detect the infeasibility of a problem.

Lemma 1. *If there is any variable x_i with $i \in [n]$ such that $a_i > 0$ and $l_i = \infty$, or such that $a_i < 0$ and $u_i = \infty$, then the problem is feasible.*

We define the concept of 'efficiency' which can tell us how valuable it is relative to the amount of capacity it uses up in the knapsack constraints.

potentiator	$(a_k \leq 0, c_k > 0, u_k = +\infty)$ or $(a_k \geq 0, c_k < 0, l_k = -\infty)$
accumulator	$(a_k < 0, c_k = 0, u_k = +\infty)$ or $(a_k > 0, c_k = 0, l_k = -\infty)$
incrementor	$(a_k > 0, c_k > 0, u_k = +\infty)$ or $(a_k < 0, c_k < 0, l_k = -\infty)$
decrementor	$(a_k > 0, c_k \geq 0, l_k = -\infty)$ or $(a_k < 0, c_k \leq 0, u_k = -\infty)$

Table 1: Status of a MIKP

Definition 1. Consider $k \in [n]$ and define

$$e_k = \begin{cases} c_k/a_k & \text{if } a_k \neq 0, \\ +\infty & \text{if } a_k = 0 \text{ and } c_k > 0, \\ -\infty & \text{if } a_k = 0 \text{ and } c_k < 0. \end{cases} \tag{3}$$

We say that e_k is the efficiency of variable x_k.

Furthermore, we also define some status that use them to claim the situation of our problem. Actually, we say that
These four definition are very useful, actually, we can obtain the following lemma

Lemma 2. *If MIKB is feasible and admits a potentiator, then MIKP is unbounded.*

Lemma 3. *If MIKB is feasible, and admits an incrementor x_i and a decrementor x_j such that $e_i > e_j$, then MIKP is unbounded.*

Proposition 1. *MIKP is unbounded if and only one of the following conditions hold,*

- *MIKP is feasible and admits a potentiator x_j.*

- *MIKP is feasible and admits an incrementor x_i and a decrementor x_j such that $e_i > e_j$.*

Note that even if MIKP is bounded, it may still admit an accumulator. We further define the 'triviality' of MIKP.

Definition 2. Consider an instance of MIKP which is feasible and not unbounded. If MIKP has an accumulator, we say that MIKP is trivial.

Actually, a trivial MIKP can be easily solved by considering the coefficients of the problem.

Proposition 2. *Assume that MIKP is feasible and not unbounded. In addition, let j correspond to an accumulator of MIKP. For each $k \in [n]$ such that $k \neq j$ define:*

- $U_k = \begin{cases} \lfloor u_k \rfloor & \text{if } k \in I, \\ u_k & \text{otherwise.} \end{cases}$

- $L_k = \begin{cases} \lceil l_k \rceil & \text{if } k \in I, \\ l_k & \text{otherwise.} \end{cases}$

- $x_k = \begin{cases} U_k & \text{if } (c_k > 0) \text{ or } (c_k = 0 \text{ and } u_k < \infty), \\ L_k & \text{if } (c_k < 0) \text{ or } (c_k = 0 \text{ and } l_k > -\infty), \\ 0 & \text{if } c_k = 0 \text{ and } x_k \text{ is free.} \end{cases}$

In addition, with respect to $k = j$, we claim that

$$x_j = \begin{cases} \max\{\lceil -\frac{\sum_{k \neq j} a_k x_k - b}{a_j} \rceil, l_k\} & \text{if } a_j < 0, \\ \min\{\lfloor -\frac{\sum_{k \neq j} a_k x_k - b}{a_j} \rfloor, u_k\} & \text{if } a_j > 0. \end{cases} \tag{4}$$

Then, we derive that x is well-defined and corresponds to an optimal solution of MIKP.

Actually, we can build an algorithm to detect infeasibility, unbounded and find trivial solutions.

Algorithm 1 Detecting infeasibility unbounded and finding trivial solutions

---

Input: c, a, b, e, l, u
Output: $status$
1: $e^+ \leftarrow -\infty; e^- \leftarrow +\infty$
2: **for** $i = 1$ **to** n **do**
3:     **if** $a_i > 0$ **and** $l_i = -\infty$ **or** $a_i < 0$ **and** $u_i = \infty$ **then**
4:         $status \leftarrow$ infeaible
5:     **end if**
6:     **if** x_i is a potentiator **then**
7:         $status \leftarrow$ potentiator
8:         **return**
9:     **else if** x_i is an incrementor and $e_i > e^+$ **then**
10:        $e^+ \leftarrow e_i$
11:     **else if** x_i is an decrementor and $e_i < e^-$ **then**
12:        $e^- \leftarrow e_i$
13:     **end if**
14: **end for**
15: **if** $e^+ > e^-$ **then**
16:     $status \leftarrow$ incrementor/decrementor pair
17:     **return**
18: **else if** $e^- = 0$ **then**
19:     $status \leftarrow$ accumulator
20:     A solution x is given.
21: **end if**
22: **return**

2.2 Preprocessing an instance of MIKP

In this section we are concerned with reducing an instance of MIKP to another, equivalent instance of MIKP which is easier to solve. A series of procedures for pre-processing an instance of MIKP are now presented. For a thorough introduction to preprocessing see [3].

Test if MIKP is infeasible, trival or unbounded. Using Algorithm [1] to test if MIKP is infeasible, trival or unbounded. If MIKP is feasible, not trivial and not unbounded, which means it has no potentiators and no accumulators. In addition if variable x_i is an incrementor, and x_j a decrementor, then $e_i e_j$.

Strength bound. We give a strength bound for our problem. First, we define

- $U_k = \begin{cases} +\infty & \text{if } a_k \leq 0, \\ (b - \sum_{i \neq k, a_i > 0} a_i l_i - \sum_{i \neq k, a_i < 0} a_i u_i)/a_k & \text{otherwise.} \end{cases}$

- $L_k = \begin{cases} -\infty & \text{if } a_k \geq 0 \\ (b - \sum_{i \neq k, a_i > 0} a_i u_i - \sum_{i \neq k, a_i < 0} a_i l_i)/a_k & \text{otherwise.} \end{cases}$

Then, we redefine the stronger bounds of our problem

$$\begin{cases} u_k = \min\{u_k, U_k\}, l_k = \max\{l_k, L_k\} & \text{if } k \notin I, \\ u_k = \min\{\lfloor u_k \rfloor, \lfloor U_k \rfloor\}, l_k = \max\{\lceil l_k \rceil, \lceil L_k \rceil\} & \text{if } k \notin I. \end{cases} \tag{5}$$

Fix values of variable. For a given variable x_k, we claim

$$x_k = \begin{cases} u_k & \text{if } a_k \leq 0 \text{ and } c_k \geq 0, \\ l_k & \text{if } a_k \geq 0 \text{ and } c_k \leq 0. \end{cases} \tag{6}$$

After fixing variables as described above, we can substitute out the values in MIKP and obtain a smaller problem with a new right-hand side. In the smaller problem, each variable x_k satisfies either $(a_k > 0, c_k > 0)$ or $a_k < 0, c_k < 0$.

Complement variables. For simplicity, we introduce new variable to make sure the lower-bound is always non-negative. Consider a variable x_k, and then we set

$$x_k := \begin{cases} x_k - l_k & \text{if } -\infty < l_k < 0, \\ u_k - x_k & \text{if } l_k = -\infty. \end{cases} \tag{7}$$

Sort data. Sort the variables in order of decreasing efficiency. Break first ties if variables are of integer type or not. Break second ties by value of a_k.

Aggregate variables. For any given two variables x_i and x_j, $i, j \in I$, if $a_i = a_j$, $c_i = c_j$. We aggregate these two variables into a single variable x_k such that $a_k = a_i, c_k = c_i, l_k = l_i + l_j, u_k = u_i + u_j$ and $k \in I$. This procedure will be very helpful later in spending up the branch and bound algorithm.

After the several steps, our finally propose is to reformulate the original MIKP to the following PP-MIKP

$$\max \quad \sum_{k \in P \cup N} c_k x_k \tag{8}$$

$$s.t. \quad \sum_{k \in P \cup N} a_k x_k \leq b$$

$$l_k \leq x_k \leq u_k, \forall k \in P \cup N$$

$$x_k \in \mathbb{Z}, \forall k \in I.$$

where $P = \{k : c_k > 0 \text{ and } a_k > 0\}$ and $N = \{k : c_k < 0 \text{ and } a_k < 0\}$. The PP-MIKP satisfies the following conditions:

- PP-MIKP is feasible.

- PP-MIKP is not unbounded, and is not trivial.

- The variable indices are sorted by efficiency.

- All variables x_k are such that $(a_k > 0 \text{ and } c_k > 0)$ or $(a_k < 0 \text{ and } c_k < 0)$.

- For each $k \in P \cup N$, we have $l_k 0$.

- For each $k \in P \cup N$, all finite bounds are tight; that is, there exists a feasible solution to MIKP which achieves the bound.

- There are no two identical variables.

2.3 Solving the LP relaxation of PP-MIKP

In this section, we discuss how to solve the linear programming relaxation of problem (8). That is, the problem LP-PP-MIKP.

$$\max \quad \sum_{k \in P \cup N} c_k x_k \tag{9}$$
$$s.t. \quad \sum_{k \in P \cup N} a_k x_k \leq b$$
$$l_k \leq x_k \leq u_k, \forall k \in P \cup N$$

Note that (9) is nothing more than a linear programming problem. As thus, any Simplex-based linear programming software package would do to solve it. Goycoolea presented a Simplex-liked algorithm, which extends in a simple way Dantzigs algorithm for solving the linear programming relaxation of bounded, positive coefficient knapsack problems.

2.3.1 Phase I Algorithm

Definition 3. We say that $x^* \in \mathbb{R}^{|P|+|N|}$ is tight for (8) if,

$$\sum_{k \in P \cup N} a_k x_k^* = b. \tag{10}$$

Definition 4. $x^* \in \mathbb{R}^{|P|+|N|}$ is $k - efficient$ for (9) if $l_k \leq x_k^* \leq u_k$ and

- $i \in P$ and $i > k$ implies $x_i^* = l_i$.

- $i \in P$ and $i < k$ implies $x_i^* = u_i$.

- $i \in N$ and $i > k$ implies $x_i^* = u_i$.

- $i \in N$ and $i < k$ implies $x_i^* = l_i$.

It can be easy to show that if there exists a tight feasible solution for (9), then there exists a tight optimal solution for (9). If x^* is tight and k-efficient for (8), then x^* is an optimal solution of (9).

The Phase I Algorithm takes as input an instance of (9), and does one of two things: (a) It proves that the instance is infeasible, or (b) it generates x, a k-efficient solution of the instance, having non-negative slack. The algorithm begins by defining $k = \max\{j \in P \cup U : j \in N \text{ and } u_j = +\infty\}$, assuming that if the latter set is empty, then $k = -1$. If $k = -1$, it generates the solution x, where

$$x_j := \begin{cases} l_j & \text{if } j \in P, \\ u_j & \text{if } j \in N. \end{cases} \tag{11}$$

If $k > 1$ it generates the solution x, for $j \in (P \cup N) \setminus \{k\}$,

$$x_j := \begin{cases} u_j & \text{if } j \in P \text{ and } j < k, \\ l_j & \text{if } j \in P \text{ and } j > k, \\ u_j & \text{if } j \in N \text{ and } j > k, \\ l_j & \text{if } j \in N \text{ and } j < k, \end{cases} \tag{12}$$

and
$$x_k = \max\left\{ -\frac{1}{a_k}\left(\sum_{j\neq k} a_j x_j - b, l_k\right)\right\}.$$

When $k = -1$, then the algorithm may generate an infeasible solution. In this case it is easy to see that the problem itself is infeasible. Also note that when the algorithm generates a feasible solution, this solution will be efficient. Thus, if the solution is tight it will be optimal.

2.3.2 Phase II Algorithm

The primal phase II algorithm takes as input an efficient solution of LP-PP-MIKP with non-negative slack, and finds from this an optimal solution to the problem. For this, it works by either increasing the values of positive coefficient variables, or decreasing the values of negative coefficient variables in a successive manner until the tightness condition is met, or until all of the variables are at their bounds and the iteration cant proceed.

Algorithm 2 Primal Phase II Algorithm

Input: $c, a, l, u, P, N, k, x_k, objective, activity.$
Output: $k, x_k, objective, activity, status.$
1: **while** $activity < b$ **do**
2:    **if** $k \in P$ **then**
3:       **if** $u_k < +\infty$ and $a_k(u_k - x_k) < (b - activity)$ **then**
4:          $activity \leftarrow activity + a_k(u_k - x_k)$
5:          $objective \leftarrow objective + c_k(u_k - x_k)$
6:       **else**
7:          $objective \leftarrow objective + (b - objective)c_k/a_k$
8:          $x_k \leftarrow x_k + (b - objective)/a_k$
9:          $activity \leftarrow b$
10:         $status \leftarrow$ tight optimal
11:         **return**
12:       **end if**
13:    **else**
14:       **if** $|a_k|(u_k - x_k) < (b - activity)$ **then**
15:          $activity \leftarrow activity + |a_k|(u_k - x_k)$
16:          $objective \leftarrow objective + |c_k|(u_k - x_k)$
17:       **else**
18:          $objective \leftarrow objective + (b - objective)|c_k|/|a_k|$
19:          $x_k \leftarrow x_k - (b - objective)/|a_k|$
20:         $activity \leftarrow b$
21:         $status \leftarrow$ tight optimal
22:         **return**
23:       **end if**
24:    **end if**
25:    $k \leftarrow k + 1$
26: **end while**
27: $status \leftarrow$ optimal
28: **return**

In order to preserve efficiency at each step it iterates by modifying the variables in order of decreasing efficiency. The algorithm always terminates with an optimal solution of the problem.

Algorithm (2) shows how a Primal Phase II procedure for LP-PP-MIKP may be implemented.

2.3.3 Dual Phase II Algorithm

The dual phase II algorithm takes as input an efficient solution of LP-PP-MIKP with nonpositive slack, and finds from this, an optimal solution to the problem. For this, it works by either decreasing the values of positive coefficient variables, or increasing the values of negative coefficient variables in a successive manner until the tightness condition is met, or until all of the variables are at their bounds and the iteration cannot proceed. In order to preserve efficiency at each step it iterates by modifying the variables in order of increasing efficiency. The algorithm either terminates with an optimal solution of the problem, or a proof that the problem is infeasible. Algorithm (3) shows how a Dual Phase II procedure for LP-PP-MIKP may be implemented.

Algorithm 3 Primal Phase II Algorithm

Input: $c, a, l, u, P, N, k, x_k, objective, activity.$
Output: $k, x_k, objective, activity, status.$

1: **while** $activity > b$ **do**
2:     **if** $k \in P$ **then**
3:         **if** $a_k(u_k - x_k) < (activity - b)$ **then**
4:             $activity \leftarrow activity - a_k(x_k - l_k)$
5:             $objective \leftarrow objective - c_k(x_k - l_k)$
6:         **else**
7:             $objective \leftarrow objective - (objective - b)c_k/a_k$
8:             $x_k \leftarrow x_k - (objective - b)/a_k$
9:             $activity \leftarrow b$
10:             $status \leftarrow$ tight optimal
11:             **return**
12:         **end if**
13:     **else**
14:         **if** $u_k < +\infty$ and $|a_k|(u_k - x_k) < (activity - b)$ **then**
15:             $activity \leftarrow activity - |a_k|(x_k - l_k)$
16:             $objective \leftarrow objective - |c_k|(x_k - l_k)$
17:         **else**
18:             $objective \leftarrow objective - (objective - b)|c_k|/|a_k|$
19:             $x_k \leftarrow x_k + (objective - b)/|a_k|$
20:             $activity \leftarrow b$
21:             $status \leftarrow$ tight optimal
22:             **return**
23:         **end if**
24:     **end if**
25:     $k \leftarrow k - 1$
26: **end while**
27: $status \leftarrow$ optimal
28: **return**

2.3.4 A branch and bound algorithm for MIKP

At each step of the branch-and-bound algorithm, we are presented with an instance of PP-MIKP, and an optimal solution to the corresponding linear relaxation, LP-PP-MIKP. This solution is both tight and efficient, and is fully characterized by the values $k, x_k, activity$ and $objective$. If variable k is of continuous type, we know that the current solution is optimal for PP-MIKP. Likewise, if variable k is of integer type, and in addition, x_k takes on an integer value, then the current solution is also optimal for PP-MIKP.

Now, assume that k is of integer type and x_k currently takes on a fractional value, say f_k. The simplest branching rule consists in defining two branches: In the first (which we call the down direction), we change the upper bound of variable x_k so that $x_k \lceil f_k \rceil$. In the other, (which we call the up direction), we change the lower bound of variable x_k so that $x_k \lfloor f_k \rfloor$.

After branching we would like to avoid solving the new linear programming relaxation of PP-MIKP to optimality from scratch; rather, we would prefer to hot-start the solve from the current solution that we have. This is very easy to do. In the down-direction we round down x_k, and in the up-direction we round up x_k. Then, activity and objective are updated. If activity increased, we now have an efficient solution with negative slack, so we use the dual phase II algorithm to re-solve. Otherwise, we have an efficient solution with positive slack, so we use the primal phase II algorithm to re-solve.

2.4 Improving: Using Domination property

As will be seen later in the computational results section, using a simple branch and bound algorithm as described in the previous sections is not enough to successfully tackle problems of practical size. In this section we concentrate on using a property called domination to improve the performance of branch and bound algorithms. The main idea of what will be done is that every time a variable bound is changed, this may have implications that lead us to change other bounds as well. By carefully identifying such implications, it is often possible to fix not just one, but several bounds at each node of the branch and bound tree.

2.4.1 Cost-Domination

Definition 5. Consider x^1 and x^2, two feasible solutions of PP-MIKP, x^1 cost dominates x^2 if,

$$cx^1 > cx^2 \text{ and } ax^1 \leq ax^2. \tag{13}$$

It is easy to see that a necessary condition for x to be optimal is that it is not cost-dominated by any other solution. Let us begin by giving some simple sufficient conditions for cost-domination.

Definition 6. Consider indices $i, j \in I$, and non-zero integers k_i, k_j. If,

$$a_i k_i + a_j k_j \geq 0 \text{ and } c_i k_i + c_j k_j < 0 \tag{14}$$

and in addition,

$$l_i - u_i \leq k_i \leq u_i - l_i \text{ and } l_j - u_j \leq k_j \leq u_j - l_j \tag{15}$$

we say that (i, j, k_i, k_j) define an integer cost-domination tuple.

2.4.2   An Example: Using Domination Tables

During our study of Marcos's paper, we use the following example to see how it works, which helps us a lot for understanding how domination helps us to get solution faster.

Consider the following trivial integer knapsack problem:

$$\begin{aligned}
\max \quad & x_1 - 2x_2 & (16)\\
s.t. \quad & x_1 - 2x_2 \leq 1.5\\
& x_1, x_2 \in \mathbb{Z}\\
& x_1 \geq 0, x_2 \geq 0
\end{aligned}$$

It is easy to see that the solutions given by $x_1 = 2k+1$ and $x_2 = k$ are optimal for all positive integers k, and that the optimal solution value is 1.

Observe that for $k_1 = 2, k_2 = 1$ the tuple $(1, 2, k_1, k_2)$ is lexicographically dominating. Thus, a domination-based branch and bound algorithm will solve this problem in five nodes. In fact, the optimal solution that will be obtained by applying the afore-described LP relaxation algorithm will be $x_1 = 1.5$ and $x_2 = 0$. In the down-branch, we will impose $x_1 \leq 1$, which will give us the optimal integer solution to the problem. Then, in the up- branch we will impose $x_1 \geq 2$. However, because of the domination-tuple, in this branch we will also impose $x_2 \leq 1$. This will result in an optimal value of $x_1 = 2$ and $x_2 = 0.25$. However, after branching but once more it easy to see that the branching will conclude (setting $x_2 = 0$ results in an infeasible node, and setting $x_2 = 1$ results in the optimal solution once again).

Now, note that if we modify the lower bounds so that $x_1 \geq r_1$ and $x_2 \geq r_2$, where $r_1, r_2 \in \mathbb{Z}_+$, then, the LP-relaxation of the problem has value 1.5. This means that a normal branch and bound algorithm will never terminate! In fact, it is easy to see that if we branch normally, there will always be a node in which only the lower bounds of x_1 and x_2 have been changed. This node will never be pruned, because the upper bound it provides is always 1.5.

3   Experiment results

In this section, computational results are stated to show the efficiency of this improved branch and bound method. All of the implementations were written in python, and compiled on polyps server under a Linux operating system. All of the knapsack algorithms were implemented with templates, thus allowing for them to use different types of numerical precision. However, in this study we just focus on using the standard floating point arithmetic implemented in python.

We refer to our algorithm in this project as KBB, for Knapsack Branch and Bound. All of the techniques in this chapter have been implemented, with the exception of the domination-induced reduced-cost bound improvement technique which is a later development. The KBB algorithm, by default, is set to work with a precision of 10^{-6}.

In order to assess the performance of our algorithms we use PuLP and set all parameters to their default values. All tests in this chapter will be done on a library of problems which we call knap_instance consisting of 549 problems. More than one half of the instances are $0 - 1$ knapsack problems.

In order to summarize results we shall present some tables. In these tables we make use of *time* to represent the amount of time, in seconds, to solve the problem using branch and bound. Note that we will also compare the effect of pre-precessing in time consuming, and we denote this time by *pptime*. *nvars* stands for the total number of variables in the original instance. Also we use *nppvars* to represent the total number of variables in the pre-processing instance.

nvar	4	10	15	20	22	25
kbbtime	0.009	0.116	0.468	0.737	0.966	1.118
pulptime	0.779	0.940	1.060	1.234	1.297	1.398
nvar	27	30	32	35	37	40
kbbtime	1.576	2.129	2.776	3.357	4.095	4.835
pulptime	1.430	1.519	1.565	1.628	1.788	1.813

Table 2: Comparison of KBB and pulp

In order to present additional information we will also use a graph to show comparison, the time will be plotted in the x-axis, and the number of instances in the y-axis. In this way, a point (x, y) on a curve will mean that for a 50 x-size instances, the implement spend y. These curves provide a wealth of information that is often missed when just looking at averages. Full tables are not presented due to the large number of problems being considered.
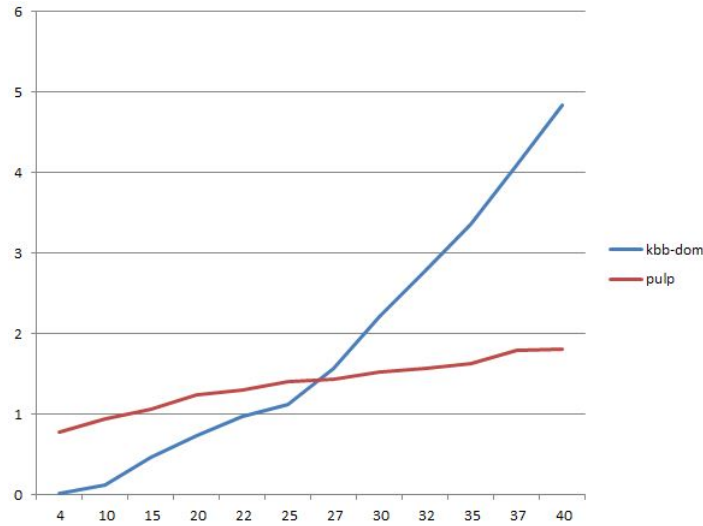

Figure 1: Comparison of KBB and pulp

As we can see from the Table 2 and Figure 1, KBB algorithm is faster than Pulp in small instances, but seems to be rather difficult to solve larger instance. This result is different from what is recorded in [2], which says KBB always outperforms PuLP, no matter how large the instance is. There are two possible reasons to this difference. First, KBB is a problem-specific algorithm, which solves complicated integer knapsack problem. However, the instances we have are only binary integer knapsack problems, which are more simple instances. Solving these instances may lose the advantage of KBB. When we emailed Dr. Goycoolea, we asked him whether he could provide us those instances he used, but got no response for this request. Second, our implementation can be improved, probability in branch and bound algorithm. Since we don't know what specific technique that the authors used in his implementation, we only adopt Depth-First-Search strategy, the criteria by which to select the node down which to dive was not clear. We guess the latter reason is the main one to cause this difference between our implementation and the original one.

Second, we try to figure out what it is that makes KBB work so much better than PuLP, we show first that part of this difference comes from the pre-processing. In this part, due to the

difficulty of finding the appropriate instances, we randomly generate some more general integer knapsack instances of dimension from 4 to 50. For problems in each dimension, we generate 20 instances. All the weight vectors and cost vectors have integer entries from -100 to 100. We compare the number of variables before and after pre-processing, computing time as following.

nvar	4	10	15	20	25	30	40	50
nppvar	2	4.9	8.1	10.2	12.4	14.5	18.7	26.2
time	0.004	0.033	0.245	0.493	0.988	1.503	2.936	4.706
pptime	0.014	0.027	0.078	0.169	0.381	0.469	1.168	2.986
pulptime	0.342	0.403	0.457	0.514	0.557	0.599	0.686	0.702
pppulptime	-	-	0.397	0.431	0.483	0.482	0.503	0.612

Table 3: The effectiveness of preprocessor

In table 3, *nvar* represents the number of variables before pre-processing, *nppvar* represents the average number of variables after pre-processing. We find an interesting things, the average number of variables always reduces around half after pre-processing. The reason of this might be that this 6 steps pre-processing procedure we mentioned in section 2, which leads to half of the variables become redundant after fixing values of variable and aggregating variables.
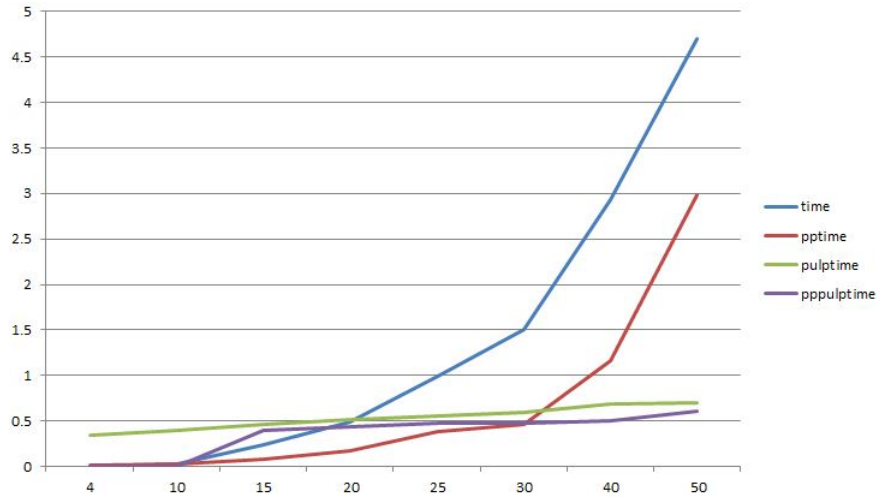

Figure 2: The effectiveness of preprocessor: computing time

In terms of computing time, *time* means the computing time for KBB algorithm without pre-processing procedure to solve instances, while *pptime* means the the computing time of KBB algorithm with pre-processing. *pulptime* and *pppulptime* represent the time of PuLP algorithm without and with pre-processing respectively. However, when the number of variables are 4 and 10, it throws out Error if we use this pre-processor on PuLP, which is surprising. We find that this pre-processor has more impact on KBB algorithm than PuLP algorithm. This is not surprising, perhaps, if we consider that CPLEX already counts with its own, very effective set of pre-processing routines. On the more difficult instances pre-processor provides much time reduction which is also as our expectation.

Then, we show that under the same pre-processing, what will happen if we didn't use domination based branching strategy. Table 4 and Figure 3 give us the comparison of computing

time between PuLP, KBB with domination and KBB without domination, represented by pulp, kbb-dom and kbb-no-dom.

nvar	4	10	15	20	22	25
kbb-no-dom	0.014	0.198	0.685	1.242	1.497	1.647
kbb-dom	0.009	0.116	0.468	0.737	0.966	1.118
pulp	0.779	0.940	1.060	1.234	1.297	1.398
nvar	27	30	32	35	37	40
kbb-no-dom	2.051	2.353	2.919	3.493	4.012	4.550
kbb-dom	1.576	2.129	2.776	3.357	4.095	4.835
pulp	1.430	1.519	1.565	1.628	1.788	1.813

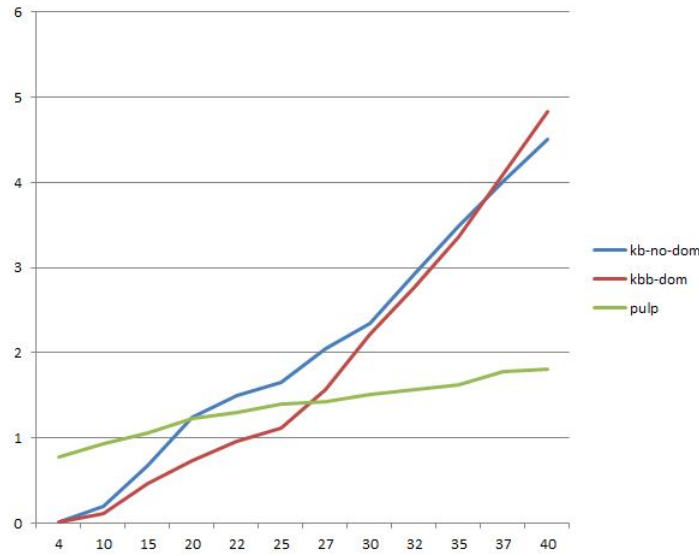Table 4: Comparison of KBB and pulp

Figure 3: Comparison of KBB and pulp

This comparison shows the effectiveness of domination strategy also depends on the dimension of instances. When the instances have small number of variables, domination strategy may lead to a huge improvement to KBB algorithm. However, as the dimension of problems grows, the time gap between KBB with domination and KBB without domination strategy becomes smaller and smaller. Then dealing with instances with 37 and 40 variables, it is even faster if we give up domination, which is different from what was recorded by Goycoolea.

Again, the problem seems come from the implementation. Domination strategy uses a large number of iterations to get a tigher bound. This number of iterations dramatically increases with the number of variables in our implementation, which can be hardly improved by us. Thus, even though the tighter bound leads to the less iterations in branch and bound, the cost of which also cannot be ignored. However, as in [2], domination has a greater chance to result in an improvement, which means that there should be some ways to improve our implementation to domination algorithm.

Finally, we change the parameter *branch direction* to see what happens. As in our first experiment shows, we adopt a Depth-First-Search strategy, but the criteria by which to select

the node down which to dive was not clear. Here we set parameter *branch direction* to be *up* and *down*. The instances we consider here are also randomly generated instances whose dimension is 30. Actually, we found that if we use *down direction*, the average time to solve those instances is $1.923s$, while if we choose *up direction*, the time turns to be $2.219s$.

We find the two diving strategies performed similarly well when considering the entire test set. This does not mean, however, that the instances performed similarly well on a one-to-one basis. In fact, it was observed that some particular instances were very sensitive to the diving strategy employed. This is partially show from the avg time since it can be seen how the average time varied with the diving strategies.

4 Summary

In this project, We implemented the algorithm in [2]. With seven subroutine algorithms and a main function. We list the function of these seven functions as following

- By introducing the definition of efficiency, the first algorithm show us it the original is infeasible ,unbounded or trivial solution. In our code, it named pre-processing.

- After the first step, if we detected that the original problem is not trivial, then we further use a simple eight-step pre-processing procedure in order to reduce an instance of MIKP to another, equivalent instance of MIKP, so called PP-MIKP which is easier to solve.

- We then focus on solving the LP relaxation of PP-MIKP which derived from the second step. First we the algorithm to solve a so-called Primal phase I algorithm, which can verify whether an instance is infeasible or obtain x, a k-efficient solutions with non-negative slack.

- Move on to this so-called primal phase II algorithm, which takes the k-efficient solution from the primal phase I algorithm and supply an optimal solution to the LP relaxation problem or show the infeasibility of the LP relaxation problem .

- We can also use dual phase II algorithm to derive the optimal solution or the LP relaxation or show if this problem is infeasible.

- Next a branching algorithm is provided, which is a easy way to avoid solving the new LP relaxation of PP-MIKP to optimality from scratch. By using the down direction and upper direction, we update activity and objective, and then choose primal phase II or dual phase II to resolve the problem.

- The last algorithm is what we used to speed up the branching and improve the bound, including cost-Domination, lexicographic-Domination. Furthermore, by using the domination, we improve the branch and bound algorithm, by building a domination table.

After implementing it, we conducted extensive computational experiments. We aim to reproduce the improvement brought by the new algorithm KBB brings in [2]. In the practical test, we use 549 binary knapsack instances to derive the numerical results, because we can't find the same instance the author used in their numerical experiment. What's more, instead of comparing the KBB algorithm with Cplex solver, our algorithm is the comparison with KBB algorithm and Pulp solver. Implemented in python, we showed that the KBB is more efficient than pulp when solving the small size problem, However, when solving the large size problem, the efficiency decreased. We assume the difference comes from the language we used, since in [2], the authors implement algorithm in C++. After this, we generated some random problem

and further show the performance of pre-processing and simplification of our algorithm and stated all the result in the table 2 and figure 2. At last, we show the importance of domination that is used to improve the efficiency of branch and bound algorithm.

References

[1] ILOG CPLEX. High-performance software for mathematical programming and optimization, 2005.

[2] Ricardo Fukasawa and Marcos Goycoolea. On the exact separation of mixed integer knapsack cuts. *Mathematical programming*, 128(1-2):19–41, 2011.

[3] Martin WP Savelsbergh. Preprocessing and probing techniques for mixed integer programming problems. *ORSA Journal on Computing*, 6(4):445–454, 1994.