Improving Branch-And-Price Algorithms For Solving One Dimensional Cutting Stock Problem

Report
of
M. Tech. Project
Stage-II

by

Soumitra Pal
Roll No: 05305015

under the guidance of

Prof. A. G. Ranade
Computer Science and Engineering
IIT Bombay

Department of Computer Science and Engineering
Indian Institute of Technology, Bombay
Mumbai
Acknowledgments

I would like to thank my guide, Prof. A. G. Ranade, for his consistent motivation and directions throughout this work.

Soumitra Pal
MTech-I
CSE, IIT Bombay
Abstract

Column Generation is a technique for solving linear programming problems with larger number of variables or columns. This technique was first applied to large real life cutting stock problem by Gilmore and Gomory. Since then several researchers have applied the column generation technique to many real life applications. Subsequently the technique is combined with the standard branch-and-bound method to solve integer programming problems with huge number of variables. This combined method is commonly known as branch-and-price.

In this report, we discuss the work undertaken in our project on how to improve the branch-and-price method to solve industrial 1D cutting stock problem which requires integer programming formulation with huge number of columns. In the first stage and the initial part of second stage, we explored the details of branch-and-price method. In the second stage we experimented to improve one subtask of the branch-and-price. We provide an outline of the next stage of our work.
## Contents

Acknowledgements iii

Abstract v

1 Introduction 1
   1.1 Cutting stock problem ................................................. 1
   1.2 Gilmore-Gomory formulation ........................................... 2
   1.3 Column generation approach .......................................... 3
   1.4 Branch-and-price ..................................................... 4
   1.5 Organization of the report ........................................... 4
      1.5.1 Work done in stage 1 ........................................ 4
      1.5.2 Work done in stage 2 ........................................ 4
      1.5.3 Work planned for stage 3 ..................................... 5

2 Work done in first stage 6
   2.1 Generic branch-and-price algorithm ................................... 6
   2.2 Branching .............................................................. 7
      2.2.1 Conventional branching on a single variable .................... 7
      2.2.2 Problem in conventional branching ................................ 8
      2.2.3 Branching in binary cutting stock problem ....................... 9
      2.2.4 Branching in general cutting stock problem .................... 12
   2.3 Bounding .............................................................. 13
   2.4 Heuristics for initial set of columns ................................ 13
   2.5 Summary .............................................................. 13

3 Work done in second stage 14
   3.1 Branch-and-price in general cutting stock problem ................. 14
      3.1.1 Vance [1998] ......................................................... 15
3.1.2 Vanderbeck [1999] .................................................. 16
3.1.3 Branching scheme .................................................. 16
3.2 Conclusion of literature survey ..................................... 18
3.3 Knapsack problem with forbidden list (KPFL) .................... 19
  3.3.1 Dynamic programming solution for KPFL ...................... 19
  3.3.2 Worstcase performance of branch & bound ...................... 20
3.4 Random Knapsack in polynomial time .............................. 20
  3.4.1 Core algorithm based proof .................................. 20
  3.4.2 Isolating lemma based proof .................................. 23
3.5 An experimentation on accelerating column generation ............ 24
  3.5.1 Formulation .................................................. 25
  3.5.2 Proof of correctness .......................................... 26
  3.5.3 COIN-OR based implementation ............................... 26
  3.5.4 Experimental results ........................................ 26
3.6 Summary .......................................................... 26

4 Work planned for third stage ......................................... 29
  4.1 Comparative study of the existing algorithms ..................... 29
  4.2 Work on getting the integer solution ............................ 29
  4.3 Summary ........................................................ 30

5 Conclusions .......................................................... 31
List of Figures

3.1 Break item and Dantzig ray .................................................. 21
List of Tables

3.1 Experimental result for the new technique of relaxing the master problem . . . . 27
Chapter 1

Introduction

Branch-and-price is a well established technique for solving large scale integer programming problems such as industrial cutting stock problem. This method, combines the standard branch-and-bound framework of solving integer programming problems with column generation. In each node of the branch-and-bound tree, the bound is calculated by solving the LP relaxation. To solve the LP relaxation, column generation technique is used. In this report we discuss our work on improving the performance of the branch-and-price based algorithms for solving the industrial one-dimensional cutting stock problem.

In this chapter, we introduce the cutting stock problem and give an overview of how the problem can be solved using branch-and-price. In section 1.1 we describe the problem. We give an integer programming formulation for solving the problem in section 1.2. As it is generally done, the first attempt should be to solve the problem by solving its LP relaxation. There are two main issues in solving the model - (i) huge number of columns in the formulation and (ii) converting the LP solution to an integer solution. In section 1.3 we discuss how the first issue could be taken care of using column generation. The second issue is solved by using branch-and-bound method which will be briefly introduced in section 1.4.

Finally, we provide an outline of the rest of the chapters in section 1.5 at the end of this chapter. In section 1.5.1 we give an overview of the work done in stage 1 of the project. We give a brief discussion of the work done in stage 2 in section 1.5.2. Finally, what we are planning to accomplish in the final stage is discussed briefly in 1.5.3.

1.1 Cutting stock problem

The cutting stock problem arises from many physical applications in industry. For example, in a paper mill, there are a number of rolls of paper of fixed width (these rolls are called *raws*), yet different manufacturers want different numbers of rolls of various-sized widths (these rolls are called *finals, items*). How should the rolls be cut so that the least amount of left-overs are wasted? Or rather, least number of rolls are cut? This turns out to be an optimization problem, or more specifically, an integer linear programming problem.

**Example** An instance of the problem. The raw rolls all have width of $W = 10$ inches. There are orders for $m = 4$ different finals of widths as follows:
1.2 Gilmore-Gomory formulation

To solve the cutting stock problem, we use the following formulation introduced by Gilmore and Gomory [1961, 1963]. The possible cutting patterns for obtaining the finals of given widths from the raw stock are enumerated beforehand. The patterns are described by the vector \((a_{ij}, \cdots, a_{ij}, \cdots, a_{mj})\) where element \(a_{ij}\) represents the number of rolls of width \(w_i\) obtained in cutting pattern \(j\). Let \(\lambda_j\) be a decision variable that designates the number of rolls to be cut according to cutting pattern \(j\).

minimize

\[
z = \sum_{j \in J} \lambda_j
\]  

subject to

\[
\sum_{j \in J} a_{ij} \lambda_j \geq b_i, \quad i = 1, 2, \cdots, m
\]

\[
\lambda_j \text{ integer and } \geq 0, \forall j \in J
\]

where \(J\) is the set of valid cutting patterns. For the given example, the valid cutting patterns are

<table>
<thead>
<tr>
<th>(i)</th>
<th>Quantity Ordered (b_i)</th>
<th>Order Width (w_i) (inches)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>9</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>79</td>
<td>5</td>
</tr>
<tr>
<td>3</td>
<td>90</td>
<td>6</td>
</tr>
<tr>
<td>4</td>
<td>27</td>
<td>9</td>
</tr>
</tbody>
</table>

As an initial attempt, the formulation is solved using the LP relaxation. However, values of \(\lambda_j\)'s (we say \(\lambda^*_j\)) may not be all integer in the optimal solution of the LP relaxation. In that case, we can round each \(\lambda^*_j\) down to the nearest integer and obtain a solution close to actual integer solution. The residual demands of finals which are not met due to rounding down can be found by brute-force.
For the given example, optimal LP solution $z = 156.7$ corresponds to $\lambda^*_1 = 27$, $\lambda^*_3 = 90$, $\lambda^*_5 = 39.5$. Rounding each $\lambda^*_j$ down to nearest integer gives value of $z = 157$ which fortunately matches with the optimal integer solution.

### 1.3 Column generation approach

However, there are difficulties which originate due to two different reasons:

- Problems commonly encountered in the paper industry may involve huge number of variables. For example, if the raw rolls are 200 in. wide and if the finals are ordered in 40 different lengths ranging from 20 in. to 80 in., then the number of different patterns may easily exceed 10 or even 100 million. In that case, the solution may not be tractable.

- Converting an optimal fractional-valued solution to an optimal integer-valued solution is not easy. Rounding the fractional values down and satisfying the residual demands, as we mentioned in the example, may not yield the optimal result. If the finals are ordered in small enough quantities, then the patterns used in the optimal integer valued solution may be quite different from those used originally in the optimal fractional valued solutions.

An ingenious way of getting around the first difficulty was suggested by Gilmore and Gomory [1961, 1963]. The trick is to work with only a few patterns at a time and to generate new patterns only when they are really needed. The rationale is that in the final solution most of the variables corresponding to the patterns would be zero anyway. This process is called delayed column generation. Here we give a brief description of the process.

In each iteration of the simplex method we look for a non-basic variable to price out and enter the basis. That is, in the pricing step, given the dual vector $\pi$ we need to find the non-basic variable corresponding to which the reduced cost component is negative. One way to find this out is to calculate the minimum of all components of the reduced cost vector and take if it is negative.

$$\arg \min \{ \bar{c}_j = c_j - \pi a_j | j \in N \}$$  (1.4)

where $a_j$ is a column of the matrix $A$.

If the columns in $A_B$ are also considered in calculating reduced cost vector, corresponding components in the reduced cost becomes 0. Thus, inclusion of the basic columns do not change the decision. The equation 1.4 can be re-written as

$$\arg \min \{ \bar{c}_j = c_j - \pi a_j | j \in J \}$$  (1.5)

An explicit search of $j$ from $J$ may be computationally impossible when $|J|$ is huge. Gilmore and Gomory showed that the searching $j$ from $J$ explicitly is not necessary. If we look carefully, it is not $j$ that we are interested. Rather we are interested in the column $a_j$ that can replace one column of the basic matrix $A_B$. In practical applications, these columns often represent combinatorial objects such as paths, patterns, sets, permutations. They follow some embedded constrains i.e. $a_j \in A$, where $A$ represents the constrained set. For example, in the cutting stock problem with Gilmore-Gomory formulation, each column represents a cutting pattern, which must satisfy the knapsack constraint - sum of widths of finals in a particular cutting pattern must not exceed the width of the raw rolls.
Thus, in practice, one works with a reasonably small subset $J' \subset J$ of columns, with a restricted master problem (RMP). Assuming that we have a feasible solution, let $\lambda$ and $\pi$ be primal and dual optimal solutions of the RMP, respectively. When columns $a_j, j \in J$, are given as elements of a set $\mathcal{A}$ and the cost coefficient $c_j$ can be computed from $a_j$, $c_j = c(a)$, then the subproblem or oracle

$$c^* = \min \left\{ c(a) - \bar{\pi}a | a \in \mathcal{A} \right\} \quad (1.6)$$

gives answer to the pricing problem.

For the cutting stock problem, $c_j = 1, \forall j \in J$ and the set $\mathcal{A}$ is given by the constraint

$$\sum_{i=1}^{m} a_{ij} w_i \leq W, \forall j \in J \quad (1.7)$$

Thus, for the cutting stock problem, the subproblem is

$$\max \sum_{i=1}^{m} \bar{\pi}_i a_i \quad (1.8)$$

such that

$$\sum_{i=1}^{m} a_i w_i \leq W \quad (1.9)$$

$$a_i \geq 0 \quad (1.10)$$

This way of starting with a basic set of columns & generating more columns as and when necessary is known as Delayed Column Generation or simply Column Generation.

### 1.4 Branch-and-price

No efficient way of handling the second difficulty described at the beginning of section 1.3 is known. However, this difficulty is solved by using a method commonly known as branch-and-price. Here the trick is to combine the delayed column generation with the standard branch-and-bound algorithm for solving integer programs. The technique is described in detail in chapter 2.

### 1.5 Organization of the report

#### 1.5.1 Work done in stage 1

In chapter 2, we report the knowledge we got from the survey of literature on branch-and-price. All the existing algorithms follow the basic branch-and-bound algorithm outlined in that chapter. We showed how the algorithms differ in the details of the implementation of the subtasks of the algorithm, mainly, branching scheme, solution of subproblem compatible with the branching scheme, bounding scheme, and setting initial feasible columns for the master problem.
1.5.2 Work done in stage 2

In chapter 3, we describe the work done in stage 2. At the first part of this stage, we completed the pending survey of literature on branch-and-price. We provide the conclusion from that survey. We identified the tasks to be undertaken to improve the branch-and-price algorithms.

One of the tasks is to find quick solutions to subproblem which is a knapsack problem. We mention the literature we found that solves the knapsack problem in polynomial time on an average.

Many a times, the knapsack problem created as part of the subproblem is associated with a list of forbidden set of items. We compute the worst case complexity of a dynamic programming solution and compare it with the worst case complexity of standard branch-and-bound algorithm for solving such problems.

Another task identified was to devise technique to accelerate the column generation process. We identified such a technique. We provide details of the technique in that chapter.

1.5.3 Work planned for stage 3

We outline the works in the next stage of our project in chapter 4. The main target is to improve on the branch-and-bound process. Also, we need to do a comparative study of the different techniques existing in the literature.

We conclude the report in chapter 5.
Chapter 2

Work done in first stage

In the first stage of the project we did a survey on the existing literature on branch-and-price. In this chapter we describe the details of the existing branch-and-price algorithms.

Even with the stronger LP relaxations, optimal solution to the LP relaxation may not necessarily be always integral. Among the standard approaches of solving integer programs using LP relaxations, the one we are interested combines column generation with the technique of branch-and-bound. Various notations have been coined for the synthesis of column generation and branch-and-bound such as branch-and-price by Barnhart et al. [2000] and IP column generation by Vanderbeck and Wolsey [1996]. However combining column generation with branch-and-bound is not straight-forward. The difficulties include devising branching rules so that the structure of the subproblem is not disturbed and handling the trailing-of-effect for which solving the LP relaxation at each node of branch and bound tree to optimality may take large number of iterations even when the solution will not be included in the final solution.

In this chapter we discuss different issues of branch-and-price. In section 2.1 we give the basic outline of the branch-and-price algorithm. In section 2.2 we discuss the different branching techniques existing in the literature.

2.1 Generic branch-and-price algorithm

We give the outline of the branch-and-price algorithm for the cutting stock problem which is a minimization problem. All existing solutions follow this generic algorithm. However they differ in the details of the individual steps.

1. Solve the problem with a heuristic solution. The incumbent is set with this value.

2. Form a branch and bound tree with a single node in it representing the entire solution set. Mark this node undiscovered and unexamined. A node is ‘discovered’ means its lower bound is already calculated. The node is ‘examined’ only if it is processed completely.

3. Choose an unexamined node in the branch and bound tree. If no examined node exists go to 6. If the node is discovered, go to 5, else go to 4.

4. (Node is undiscovered) Get a lower bound $LB$ on the solution set represented by the node. It is done by solving the LP relaxation using column generation. If the lower
bound corresponds to a feasible integer solution $X$ and $X$ is less than \textit{incumbent}, then let \textit{incumbent} $= X$. It can be noted that this solution is the optimal integer solution for the subtree under the current node. This may not be the final integer solution because there can be better integer solution on other branches not under the current subtree. Hence, mark the node only as discovered. Go to step 3.

5. (Node is discovered) Mark this node examined. If the lower bound value \textit{LB} at this node is $\geq \textit{incumbent}$, there is no need to explore this subtree and hence go to step 3. Otherwise, divide the solution set at the node into two nodes, add them to the branch and bound tree, mark both undiscovered and unexamined. Go to step 3.

6. Stop, \textit{incumbent} gives the optimal solution value.

This is a very rough outline of branch and bound. Greater work is involved in deciding the branching rules (in step 5) and how to choose an unexamined node (in step 3). We discuss them in the subsequent sections.

2.2 Branching

The challenge in formulating a branching rule is to find a scheme of separation of the feasible solution space. The scheme is applied successively at each node of the branch and bound tree to finally get the optimal solution while eliminating fractional solutions at each node. The scheme should have the following properties:

- It should exclude the current fractional solution and validly partition the solution space of the problem.

- There should be a guarantee that a feasible integer solution will be found (or infeasibility proved) after a finite number of branches.

- The branching information can be encoded in the subproblem so that the column invalidated by the branching rule is not regenerated by the subproblem.

- In addition, a branching scheme will be effective if the scheme divides the feasible set of solutions to subsets of approximately equal size.

- Also, the branching scheme should be compatible with the column generation algorithm. That is, we can still use column generation to solve the LP relaxation of the more restricted problems defined at the successor nodes. A good branching scheme should keep the master and subproblem tractable.

In the following subsections we discuss different branching schemes that are existing in literature.

2.2.1 Conventional branching on a single variable

The obvious choice for branching should be to branch on a single variable as done in in the conventional integer programming. Degraeve and Schrage [1999] used this scheme.
2.2. BRANCHING

Branching scheme

Branching is done on a single fractional variable, say \( \lambda_q = \alpha \), \( \alpha \) is fractional, by adding a simple upper bound \( \lambda_q \leq [\alpha] \) at the left branch and a simple lower bound \( \lambda_q \geq [\alpha] \) at the right branch.

Solution to subproblem

We need to take care of the effect of this branching in the corresponding subproblem. We discuss the effect on the two branches separately. The effect on the right branch is easier to handle. Since, in this case, \( \lambda_q \) for the column \( q \) is \( \geq [\alpha] \), adding the constraint is equivalent to reducing the demand vector by the column \( q \) multiplied by \([\alpha]\) and solving the residual problem. The residual problem can be solved as if it is a new problem. We do not need to modify the subproblem at all. If the same column \( q \) appears in the solution of the residual problem with a value \( \beta \), we make \( \lambda_q = [\alpha] + \beta \) in the final solution.

The problem comes in the left branch where an upper bound on the decision variable is added. Since the variable is forcefully subdued, it is quite possible that the column corresponding to the variable will price out favourably again, using the dual prices of the new restricted master problem. However, as this column is already existing, the subproblem should not regenerate this forbidden column.

To solve this subproblem with new complication, Degraeve and Schrage [1999] made slight modification to Gilmore Gomory implicit enumeration procedure. In addition to the dual prices, demands and raw material lengths, a list of forbidden columns is also passed to this subroutine. Each time the subroutine discovers a new solution, before updating the incumbent, it searches the solution in the list of forbidden columns. It updates incumbent only if the solution is not in the list.

Degraeve and Schrage [1999] also gave an outline of the solution to the subproblem when it is an arbitrary integer program. When the variables \( x_i \) in subproblem are 0/1 variables, to prevent a column for which the set of subproblem variables at 0 is \( S_0 \) and the set of variables at 1 is \( S_1 \), following constraint is added to subproblem.

\[
\sum_{i \in S_1} x_i - \sum_{i \in S_0} x_i \leq |S_1| - 1 \tag{2.1}
\]

Arbitrary integer variables can be represented in the subproblem by representing each general integer variable by a binary expansion of 0/1 variables.

Node selection

Degraeve and Schrage [1999] used depth-first search for the branch and bound.

2.2.2 Problem in conventional branching

The problem of conventional branching using single variable is that the branching rule does not divide the set of feasible solutions in to two subsets of approximately equal size. Here, the division of feasible solutions can be thought of equivalent to dividing of the set of new columns that can get generated by the subproblem. This is easier to think if we consider the Kantorovich
2.2. BRANCHING

formulation. There, the columns in Gilmore-Gomory formulation is a part of the variable vector. And hence partition of the feasible solution according to a partition of the set of columns in Gilmore-Gomory formation is a valid one.

On the right branch, the feasible space gets very restricted. This is because the variable is lower bounded. Adding the branching constraint is equivalent to reducing the demand vector by the column multiplied by the lower bound and solving the residual problem. Since the demand vector is reduced considerably, number of possible new columns that can be generated by the subproblems also gets reduced.

On the other hand, on the left branch, since the variable bound is an upper bound, we can not think of such residual problem. Only one pattern is excluded, leading to a problem that is not much more restricted that the one at the predecessor node.

Moreover, the constraint added to the subproblem is not trivial and it may destroy the structure of the subproblem and may be making exact column generation intractable as well.

2.2.3 Branching in binary cutting stock problem

The objective of the binary cutting stock problem (BCS) is to find the minimum number of rolls of length $L$ necessary to meet exactly the demand for rolls of shorter lengths $w_i$ for $i = 1, \ldots, n$. In the binary case, the demand for each length $w_i$ is 1 i.e. $b_i = 1 \forall i$.

Vance et al. [1994] devised a branch and price algorithm for the BCS. The branching rule is based on the following proposition which applies to any 0-1 matrix $(x_{ij})$ and thus to any set partitioning problem.

Proposition 2.2.1 (Ryan and Foster [1981]) If any basic solution to the master problem is fractional, then there exists rows $l$ and $m$ of the master problem such that

$$0 < \sum_{k: x_{lk}=1, x_{mk}=1} \lambda_k < 1 \quad (2.2)$$

Branching scheme

Using the above proposition, it is possible to find the pair $l$, $m$. Otherwise the problem is already integral. The branching rule corresponding to the pair $l$, $m$ is given by

$$\sum_{k: x_{lk}=1, x_{mk}=1} \lambda_k \geq 1 \quad \text{and} \quad \sum_{k: x_{lk}=1, x_{mk}=1} \lambda_k \leq 0 \quad (2.3)$$

This branching is analogous to requiring that rows $l$ and $m$ be covered by the same column on the left branch and by different columns on the right branch. Instead of adding explicit constraints to the master problem, the invalid columns are eliminated. Correspondingly the subproblem is modified in such a way that the invalid columns are not regenerated.

On the left branch, columns containing only one of $l$ and $m$ are eliminated. The new columns generated should satisfy $x_l = x_m$. This constraint is added in the subproblem. On the right branch, columns containing both $l$ and $m$ are eliminated. Such columns are prevented from regeneration by adding the constraint $x_l + x_m \leq 1$ in the subproblem. The details of solving the subproblem is provided later.
It can be seen that the branching scheme separates the solution space into subsets of approximately equal size. On the left branch only columns with both $l$ and $m$ or none are valid. On the right branch only columns with only one of both $l$ and $m$ or none are valid. These two sets are approximately of equal size unlike the case of conventional branching.

**Solution to subproblem**

We have seen that on the left branch the subproblem should satisfy $x_l = x_m$. This can be done by merging the two items $l$ and $m$ by an item of width $w_l + w_m$ and profit $\pi_l + \pi_m$. However solving the subproblem on the right branch is trickier. The constraint $x_l + x_m \leq 1$ can be interpreted as an edge between the nodes $l$ and $m$ in a graph with all items as node. The objective of the subproblem is to solve the knapsack problem such that optimal solution contains items such that at most one from each edge is picked.

Deep in the branch and bound tree there might be subproblems where constraints of both types are added. Depending on the following three cases, Vance et al. [1994] designed three different algorithms for solving the subproblem.

At nodes where no edge constraints are present in the subproblem, Horowitz and Sahni [1974] branch-and-bound algorithm (HS) was used. The algorithm is as follows.

1. The items are sorted as in the greedy heuristic i.e. according to non-increasing profit density ($\pi_i / w_i$). The algorithm is initialized with $s = 0, \hat{x}_i = 0$ for $i = 1, \ldots, n$. That means, the algorithm starts with empty set of items. It then makes several forward and backtracking move.

2. In the forward move, the item $s + 1$ is tried to be inserted in the current solution (i.e. $\hat{x}_{s+1}$ is set to 1 if it can be accommodated). After a forward move, $s$ is updated to $s = s + 1$.

   The LP upper bound corresponding to the current solution $\hat{x}$ is given by
   
   $\begin{equation}
   U = \sum_{i=1}^{s} \pi_i \hat{x}_i + \left( W - \sum_{i=1}^{s} w_i \hat{x}_i \right) \pi_{s+1} / w_{s+1}
   \end{equation}$

   If the bound $U$ is greater than the best solution known so far, a new forward move is performed. Otherwise a backtracking move follows.

3. When the last item is considered for forward move, if the current solution $\hat{x}$ is better than the best solution known so far, then it is taken as the best solution so far.

4. In a backtracking step the last inserted item is removed i.e. $\hat{x}_s = 0$ and $s = s - 1$. The LP upper bound is calculated with this new solution i.e. $\hat{x}$. If it is greater than the best known solution, a forward move follows. Otherwise a new backtracking move is made.

5. The algorithm stops when no further backtracking can be made.

When there are edge constraints at a node, two possible cases may arise - (i) the edge constraints are non-overlapping i.e. no two edge constraints share the same item and (ii) edge constraints are overlapping. If the edge constraints overlap, that is, one item appears in more than one edge constraints, Vance et al. [1994] solved the subproblem by a general IP solver.

When there is no overlapping edge constraint, Vance et al. [1994] used a modified version of the HS algorithm. For that, they used the algorithm Johnson and Padberg [1981] (JP) for calculating
an upper bound of 0-1 knapsack problem with Special Ordered Sets (SOSs). A special ordered set (of type I) is a set of variables among which at most one can be non zero (positive). When the edge constraints are non-overlapping, each edge constraint can be represented by a SOS of size 2. Other variables can be represented by SOSs of size 1. We now discuss the algorithm JP in brief and then show how the variable ordering and bounds calculated by JP is used in the modified HS algorithm.

**JP Algorithm:** The LP relaxation of the knapsack problem with special ordered sets has the following standard form (KPSOS):

\[
\begin{align*}
\text{max} & \quad \sum_{j \in L} \sum_{i \in S_j} \pi_i x_i \\
\text{subject to} & \quad \sum_{j \in L} \sum_{i \in S_j} w_i x_i \leq W \\
& \quad \sum_{i \in S_j} x_i \leq 1 \quad \forall j \in L \\
& \quad x_i \geq 0 \quad \forall i \in S_j, j \in L
\end{align*}
\]

where \( \forall i, \pi_i > 0, w_i > 0; W > 0, \) and \( |S_j| \geq 1, \forall j \in L. \) Let for \( j \in L, \)

\[
\frac{\pi_i}{w_{ij}} = \max \left\{ \frac{\pi_i}{w_i} \mid i \in S_j \right\}
\]

and the SOSs are ordered such that

\[
\frac{\pi_{i_1}}{w_{i_1}} \geq \frac{\pi_{i_2}}{w_{i_2}} \geq \cdots \geq \frac{\pi_{i_h}}{w_{i_h}}
\]

where \( h = |L|. \) The LP solution for the KPSOS is calculated as follows

1. If \( w_{i_1} \geq W \) then the solution is given by \( x_{i_1} = W/w_{i_1} \) and \( x_i = 0, \forall i \neq i_1. \) Goto 7.
2. If \( |S_1| = 1, \) set \( x_{i_1} = 1, \) remove the variable and the SOS from the problem, subtract \( w_{i_1} \) from \( W \) and goto 1.
3. RECORD the variable \( i_s \) defined by \( \pi_{i_s} = \max \{ \pi_i \mid w_i = w^* \} \) where \( w^* = \min \{ w_i \mid i \in S_1 \}. \) That is, the minimum weight item in the SOS \( S_1. \) If there are multiple minima, take the minimum with maximum profit. Replace \( S_1 \) by the set \( S_1^* = \{ i \in S_1 \mid \pi_i > \pi_{i_s} \}. \) For all \( i \in S_1^* \) do the following:
   (a) Replace \( \pi_i \) by \( \pi_i - \pi_{i_s}. \)
   (b) Replace \( w_i \) by \( w_i - w_{i_s}. \)
   Replace \( W \) by \( W - w_{i_s}. \)
4. If \( S_1^* = \emptyset, \) fix \( x_{i_s} = 1 \) and goto 1.
5. Update the order of the remaining SOSs as defined by the equation 2.10.
6. If a variable \( x_{i_k} \) from the SOS \( S_1 \) was RECORDED previously then
   (a) If \( w_{i_k} \geq W \) then set \( x_{i_1} = W/w_{i_1}, \) set \( x_{i_k} = 1 - W/w_{i_k}, \) and \( x_i = 0, \forall i \neq i_1 \neq i_k. \) Goto 7.
   (b) Keep only \( x_{i_1} \) as RECORDED and set \( x_{i_k} = 0. \) Goto 1.
7. Calculate the LP optimal value using the values of variables \( x_i \) set.
Modified HS algorithm: The modified HS algorithm proceeds as the original with the following exceptions.

- The greedy ordering of the variables is replaced by the ordering of the SOSs as defined by the equation 2.10.
- In a forward move, at most one item from each SOS is tried to be inserted in the current solution.
- The LP upper bound is calculated using the JP algorithm.

\[
U = \sum_{j=1}^{s} \sum_{i \in S_j} \pi_i \hat{x}_i + JP\{(s + 1), \ldots, h\}, W - \sum_{j=1}^{s} \sum_{i \in S_j} w_i \hat{x}_i \quad (2.11)
\]

where \(s\) denotes the index of SOS corresponding to the item inserted last and \(JP(S, W)\) returns the upper bound of the KPSOS containing the set of SOSs \(S\) and knapsack size \(W\).
- In a backtracking move, the item inserted last is removed from the current solution and other item (if any) in the corresponding is enabled for insertion if it was not considered already.

Node selection

Since the subproblem on the left branch is easier to solve, the implementation by Vance et al. [1994] always explored the left branch first. With their branching scheme, there was another place where options could be made. If there were several \(l\) and \(m\) pairs, they first searched for the pair that satisfied the knapsack constraint with equality. If no such pair existed, the largest feasible pair was chosen. The idea was to identify ‘good’ pattern with little or no waste. They also tried to avoided the overlapping edge constraints by picking pairs which do not overlap with previously used pairs.

2.2.4 Branching in general cutting stock problem

The branching scheme used by Vance et al. [1994] is based on the proposition 2.2.1. The proposition is not applicable when the matrix contains elements with general integer values. Hence, the same branching rule can not be applied here. Vance [1998] and Vanderbeck [1999] implemented branching rules for general cutting stock problem. The key idea they used is to find a set of columns \(\hat{Q}\) such that the sum of the variables corresponding those columns is some fractional value \(\alpha\). The branching is based on this fractional value. That is the branches are

\[
\sum_{q \in Q} \lambda_q \geq \lceil \alpha \rceil \quad \text{and} \quad \sum_{q \in Q} \lambda_q \leq \lfloor \alpha \rfloor \quad (2.12)
\]

The details of the branching is explored in the next stage of our work.
2.3 Bounding

One of the main problems of solving integer programming problems using branch-and-price is the tailing-off effect or slow convergence of the column generation process at each node. It takes many iterations at the end to get the optimal LP solution. This is a waste if the solution thus obtained is not included in the final solution. However, the main objective of solving the LP is to obtain a lower bound on the final integer solution. Fortunately, it is generally not necessary to solve the LP to optimality to get the lower bound. It is a consequence of the following proposition by Vance et al. [1994].

Let $z_{\min}$ be the optimal value of the LP at the root node master problem, $z_B$ be the optimal value of the root node restricted master problem over the current subset of columns and $\bar{c}_{\min}$ be the reduced cost of the column generated by the subproblem and $z_{IP}$ is the optimal IP solution.

**Proposition 2.3.1** If $\lceil z_B \rceil = \lceil z_B / (1 - \bar{c}_{\min}) \rceil$, then $\lceil z_B \rceil \leq z_{IP}$.

The proposition implies that when $\lceil z_B \rceil = \lceil z_B / (1 - \bar{c}_{\min}) \rceil$, $\lceil z_B \rceil$ is a lower bound to $z_{IP}$ and hence column generation can be stopped.

Although the above proposition is applicable to the root node only, at subsequent nodes once the LP optimal to the restricted master drops below the $\lceil z_B \rceil$ calculated at the root node, it is not necessary to continue column generation because the lower bound will not be improved further.

2.4 Heuristics for initial set of columns

To start the column generation we need a set of feasible columns. Vance et al. [1994] used the First-Fit Decreasing heuristic to obtain the initial set of feasible columns. In this heuristic, items in descending width are cut from already used rolls. If they can can not be cut, new rolls are used.

2.5 Summary

We discussed the basics of branch-and-price algorithms to solve IP problems using branch-and-bound and column generation. We gave the outline of the generic algorithm. We discussed the details of the branching rules, bounding rules and heuristics for the initial set of columns used in the algorithm by Vance et al. [1994] for solving the binary cutting stock problem.
Chapter 3

Work done in second stage

In chapter 3, we describe the work done in stage 2. At the first part of this stage, we completed the pending survey of literature on branch-and-price. We studied the branch-and-price algorithm for solving the general cutting stock problem as in Vance [1998] and Vanderbeck [1999]. We give the details in section 3.1. We provide the conclusion from that survey in 3.2 In that section we identified the tasks to be undertaken to improve the branch-and-price algorithms.

It is evident from the survey that many a times, the knapsack problem created as part of the subproblem is associated with a list of forbidden set of items. In section 3.3 we compute the worst case complexity of a dynamic programming solution and compare it with the worst case complexity of the standard branch-and-bound algorithm for solving such problems.

One of the tasks identified from the survey is to find quick solutions to subproblem which is a knapsack problem. We mention the literature we found (Beier and Vöcking [2004]) that solves the knapsack problem in polynomial time on an average. We describe that in section 3.4.

Another task identified is to devise technique to accelerate the column generation process. We identified such a technique. We provide details of the technique in section 3.5.

3.1 Branch-and-price in general cutting stock problem

In the general cutting stock problem, the demand for items can possibly be more than 1. Hence the column coefficients and right hand sides are general integers rather than 0-1. This does not change the bounding scheme. It also does not change the heuristics mentioned earlier to generate the initial set of columns as done in the binary case. However, this has an effect on the branching policy.

The branching scheme used by Vance et al. [1994] based on the proposition 2.2.1 is not applicable when the matrix contains elements with general integer values. Hence, the same branching rule can not be applied here. Here we explore the two different branching scheme for the general problem. One by Vance [1998] and other by Vanderbeck [1999]. The key idea they used is to find a set of columns $\hat{Q}$ such that the sum of the variables corresponding those columns is some fractional value $\beta$ possibly greater than 1. The branching is based on this fractional value. That is the branches are
3.1. BRANCH-AND-PRICE IN GENERAL CUTTING STOCK PROBLEM

\[ \sum_{q \in \hat{Q}} \lambda_q \geq \lceil \beta \rceil \quad \text{and} \quad \sum_{q \in \hat{Q}} \lambda_q \leq \lfloor \beta \rfloor \]  

(3.1)

We give the details in the following subsections.

3.1.1 Vance [1998]

Branching scheme

To determine the set \( \hat{Q} \), Vance [1998] used the following fact shown in Vanderbeck and Wolsey [1996]. If the solution to primal is fractional, it is possible to identify a set of rows \( S \) and integers \( \{ \alpha_l : l \in S \} \) such that

\[ \sum_{q : a_{lq} \geq \alpha_l \forall l \in S} \lambda_q = \beta_S \]  

(3.2)

is fractional. The set \( S \) is called a branching set. Then branching constraints on the two child nodes would be

\[ \sum_{q : a_{lq} \geq \alpha_l \forall l \in S} \lambda_q \geq \lceil \beta_S \rceil \quad \text{and} \quad \sum_{q : a_{lq} \geq \alpha_l \forall l \in S} \lambda_q \leq \lfloor \beta_S \rfloor \]  

(3.3)

These constraints place upper or lower bounds on the number of columns with \( a_{lq} \geq \alpha_l \) for all \( l \in S \) that can be present in the solution. However, in general, it is not possible to guarantee about the size of the set, \( |S| \).

However, the size of the set gets restricted if only maximal cutting patterns are allowed as columns in the master problem formulation. By maximal, we mean that the waste left after cutting this pattern is shorter than the length of the smallest item. Vance [1998] utilized another fact from Vanderbeck and Wolsey [1996] that in this special case, given a fractional solution \( \lambda^* \) to the LP relaxation of the master problem, any fractional column \( \lambda_q \) defines a potential branching set, letting \( S = \{ l : a_{lq} > 0 \} \) and \( \alpha_l = a_{lq} \) for all \( l \in S \). Thus, the branching decisions can be enforced by changing the the upper or lower bounds on a single variable in the master problem. This implies that the branching scheme is same as the conventional branching where only a single variable is used for branching.

Solution to the subproblem

Since the branching scheme is same as the conventional branching, it needs to solve the knapsack problem with a list of forbidden columns. However, the fact that the forbidden patterns are all maximal makes eliminating the easier than it would be for general patterns.

Vance [1998] modified the Horowitz and Sahni [1974] branch-and-bound algorithm (HS) for solving the subproblem. Since the algorithm solves 0-1 knapsack problems, a logarithmic transformation is used to convert the integer knapsack problem into an equivalent 0-1 problem.

The transformation goes as follows. For each item \( i \in \{ 1, 2, \ldots, n \} \), let the implicit upper bound (i.e. the maximum value that can be assumed by the associated column entry) be

\[ q_i^{\max} = \min \{ b_i, \left\lfloor \frac{W}{w_i} \right\rfloor \}, \]  

(3.4)
and define \( n_i = \lceil \log(q_i^{\max} + 1) \rceil \) for \( i = 1, \ldots, n \) and \( n' = \sum_{i=1}^{n} nn_i \). Then the 0-1 vector \( q' \in \{0,1\}^{n'} \) associated with feasible column \( q \in \mathcal{N}^n \) is defined by the relation \( q_i = \sum_{k=0}^{n_i-1} 2^k q_{p_i+k} \forall i \) where \( p_i = 1 + \sum_{k=1}^{i-1} n_k \). The new profit vector \( \pi' \in \mathbb{R}^{n'} \) and weight vector \( w' \in \mathbb{N}^{n'} \) for the 0-1 problem that are counterpart of \( \pi \) and \( w \) for the integer knapsack problem are \( \pi'_i = 2^k \pi_i, w'_i = 2^k w_i \) for \( l = p_i + k, k = 0, \ldots, n_i - 1 \) and \( i = 1, \ldots, n \). It can be noted that the index \( i = 1, \ldots, n \) is used to refer to the original items and the index \( l = 1, \ldots, n' \) is used to refer to the 0-1 component of the transformed knapsack problem which takes the following formulation

\[
\begin{align*}
\text{max} & \quad \sum_{l=1}^{n'} \pi_l a_l \\
\text{sub to} & \quad \sum_{l=1}^{n'} w_l a_l \leq W \\
& \quad \sum_{l=p_i}^{p_i+n_i-1} 2^{l-p_i} a_l \leq q_i^{\max} \forall i \\
& \quad a_l \in \{0,1\}
\end{align*}
\]

The solution to the subproblem thus requires to make sure the following

1. the new patterns generated are maximal
2. no forbidden pattern gets generated

The first goal is achieved by including items whose associated profit (dual variable) is zero. Thus, at the end, the algorithm continued to add the items with zero profit until it is full although they do not change the reduced cost of the resulting pattern.

The other goal is met with the following modification in the HS algorithm. For each forbidden pattern \( q \), the item with \( a_{iq} > 0 \) that appears last in the greedy ordering is marked. Actually, since the general integer knapsack has been transformed into a 0-1 knapsack problem, there may be several 0-1 variables corresponding to item \( i \). Each of these items are marked. In the forward pass of the HS algorithm, when one marked item is added, it is checked if the recent pattern is in the forbidden list. If so, the item is not added. Since, in the HS algorithm always adds items in the greedy ordering, it is enough to check for forbidden pattern only when the last item included in the pattern is considered for addition.

In summary, Vance [1998] implemented branching by using bound on a single primal variable, and correspondingly solved the modified subproblem by a modified HS algorithm in the special case when maximal patterns considered for inclusion in the primal.

3.1.2 Vanderbeck [1999]

3.1.3 Branching scheme

Vanderbeck [1999] used the following result in Vanderbeck and Wolsey [1996] for devising the branching scheme.
Proposition 3.1.1 Given a fractional solution $\lambda$ of the master problem, it is possible to find a subset of columns

\[ \hat{Q} = \{ q \in Q : q'_l = 0 \quad \forall l \in O \quad \text{and} \quad q'_l = 1 \quad \forall l \in P \} \] (3.9)

such that $\beta = \sum_{q \in \hat{Q}} \lambda_q$ is fractional and hence can be used for branching in the following form. Here $q'$ is the 0-1 representation of the column $q$ as described in the section on scheme used by Vance [1998].

\[ \sum_{q \in \hat{Q}} \lambda_q \leq \lfloor \beta \rfloor \] (3.10)

or

\[ \sum_{q \in \hat{Q}} \lambda_q \geq \lceil \beta \rceil \] (3.11)

Vanderbeck and Wolsey [1996] also provides a way to find the subsets $O$ and $P \subset \{1, \ldots, n'\}$ that completely characterize $\hat{Q}$ with $|O| + |P| \leq \lfloor \log f \rfloor + 1$ where $f$ is the sum of the fractional part of the variables in the fraction solution $\lambda$ i.e. $f = \sum_{q \in Q} (\lambda_q - \lfloor \lambda_q \rfloor)$.

At a branch and bound node $u$, after such branching constraints have been added to the master problem, the master LP takes the following form:

\[ \min \sum_{q \in Q} \lambda_q \] (3.12)

s.t. \[ \sum_{q \in Q} q_i \lambda_q \geq b_i \quad i = 1, \ldots, n \] (3.13)

\[ \sum_{q \in Q^j} \lambda_q \leq K^j \quad \forall j \in G^u \] (3.14)

\[ \sum_{q \in Q^j} \lambda_q \geq L^j \quad \forall j \in H^u \] (3.15)

\[ \lambda_q \geq 0 \quad q \in Q \] (3.16)

where $G^u$ and $H^u$ are sets of branching constraints of the form 3.10 and 3.11 respectively, that are present in the LP corresponding to node $u$. $Q^j$ denotes the corresponding set of columns defining the constraint $j$. $K^j$ or $L^j$ define the corresponding right hand side of the constraint $j$.

Solution to the subproblem

Because of the addition of extra constraints of the forms $G^u$ and $H^u$ in the master problem at the node $u$, we should also include the corresponding dual multipliers too in the subproblem.
3.2 CONCLUSION OF LITERATURE SURVEY

The subproblem takes the following form

$$\max_{\pi'} \sum_{l=1}^{n'} \pi_l a_l - \sum_{j \in G^u} \mu_j g_j + \sum_{j \in H^u} \nu_j h_j$$

subject to

$$\sum_{l=1}^{n'} w_l a_l \leq W$$

$$\sum_{l=p_i}^{p_i+n_i-1} 2^{l-p_i} a_l \leq q^\text{max}_i \quad \forall i$$

$$g_j \geq 1 - \sum_{l \in \text{lin} O^j} a_l - \sum_{l \in P^j} (1 - a_l) \quad \forall j \in G^u$$

$$h_j \leq 1 - a_l \quad \forall l \in O^j, j \in H^u$$

$$h_j \leq a_l \quad \forall l \in P^j, j \in H^u$$

$$a_l \in \{0, 1\}$$

where $$(\pi', \mu, \nu) \in R^{n'+|G^u|+|H^u|}$$ is an optimal dual multiplier to the master LP at node $u$ and $\pi'$ and $w'$ are the 0-1 equivalent of $\pi$ and $w$ respectively.

In summary, Vanderbeck [1999] uses a new technique of finding the fractional columns to branch on, adds the branching constraints to the primal, adds new variables and constraints in the subproblem accordingly and solves the subproblem as a general integer programming problem.

3.2 Conclusion of literature survey

From the survey, we got the knowledge that the performance of a branch-and-price based algorithm is dependent on the following main tasks of the algorithm. Performance of the algorithm could be improved by improving one or more of these tasks.

1. Solving the LP relaxation at each node. It depends on the following two subtasks
   
   (a) Solving the restricted master problem. Here, initially, we take a very restricted or constrained form of the master problem and gradually improve the solution by somehow relaxing the formulation by adding new columns. This is continued until we get an optimal solution. Any trickier formulation which can accelerate the finding of the optimal set of columns will improve the overall performance of the algorithm.

   (b) Solving the subproblem. Here, we need to consider two aspects. In a single node, as the column generation progresses, the knapsack problem to be solved becomes harder because of the improved dual variables. We need to see how to solve the subproblem efficiently in this case. Also, as we go deeper in to the branch and bound tree, the subproblem becomes harder. We need to take this aspect also in consideration.

2. Finding better branching scheme with compatible subproblems. We have to make sure that the branching scheme divides the solution space in to two parts of approximately same size. Also, the scheme should be such that it does not make the subproblem too hard to solve.

With the above observation we started exploring few things in the above mentioned areas which are described in the subsequent sections.
3.3 Knapsack problem with forbidden list (KPFL)

We have seen that if we use the conventional branching scheme where a single variable is used for branching, the subproblem takes the form of a knapsack problem with a list of forbidden patterns as an extra parameter. As a first step, we tried to find a dynamic programming solution to this problem and compare the worstcase performance against the basic branch and bound way of solving the problem.

3.3.1 Dynamic programming solution for KPFL

We have the following dynamic programming solution to the KPFL. For simplicity, we do not show the table bases implementation of the dynamic programming. The implementation with dynamic programming table is a straight forward conversion of the following recursive algorithm.

Inputs: 
- \( n \) = number of items
- \( w \) = set of weights of the items
- \( p \) = set of profits for the items
- \( W \) = knapsack capacity
- \( L \) = list of forbidden patterns

Output: \( S \) = list of items (indices) included initially empty

KPFL\((n, w, p, W, L, S) := KPFL\)-AUX\((n, w, p, W, L, S)\)

KPFL\)-AUX\((i, w, p, W, L, S)\):
  if \((i == 0)\):
    if \((S \text{ is in } L)\)
      return \(-1\)
    else return profit of \(S\)
  else
    if \((w[i] > W)\)
      return KPFL\)-AUX\((i-1, w, p, W, L-\{patterns not having item } i, S)\)
    else
      \(m1 = KPFL\)-AUX\((i-1, w, p, W-w[i], L-\{patterns having item } i, S \cup \{i\})\)
      \(m2 = KPFL\)-AUX\((i-1, w, p, W, L-\{patterns not having item } i, S)\)
      if \((m1 == -1)\)
        return \(m2\)
      else
        if \((m2 == -1)\)
          return \(m1+p[i]\)
        else
          return \(\max\{m1+p[i], m2\}\)

Worstcase time complexity It can be seen that the auxiliary subroutine has three input parameters which differ. The parameters are \(i, W\) and \(L\). A quick calculation should reveal
that the size of the dynamic programming table is \( = O(n \times W \times 2^n) \) since the number of possible subsets, and possibly the forbidden list, of \( n \) items is \( 2^n \).

However, a deeper look into the operation on the table will reveal that only a few entries of the dynamic programming table are in fact accessed by the algorithm. For the particular value of \( i \) and \( W \) there can be at most 2 possible values of \( L \), this is because the corresponding item could be present or not present in the forbidden list. Hence, the maximum number of entries in the table that are accessed by the algorithm is \( O(2^nW) \). Again for filling each of these entries takes \( O(L) \) time. Hence, the time complexity = \( O(2^nWL) \).

### 3.3.2 Worstcase performance of branch & bound

The time complexity of the branch-and-bound algorithm is \( O(2^n + nL) \). This is because in the worst possible case the branch-and-bound tree can be the complete binary tree with height \( n \) and since at each level the list of \( L \) forbidden columns is distributed among the nodes, the time required summed up for a level is \( L \).

Hence, we see that if \( C < 2^n \), in the worst case, dynamic programming solution works better. On the other case, branch-and-bound algorithm works better.

### 3.4 Random Knapsack in polynomial time

By this time we were enthusiastically looking in literature for some algorithm which gives very quick solutions to the knapsack problems, if not always, on an average. In the mean time, we came to know about the work of Beier and Vöcking [2004]. We were excited to know about an algorithm for solving knapsack problem in polynomial time. We went through the details. Later we found another easier to understand proof of the polynomial average-case time complexity of knapsack problems. However, it will be clear at the end of the description of our small implementation, conducted in the later part of this stage of the project, that the knapsack instances generated at the time of solving the subproblems are not random and most of the time the profits and weights are correlated. Still we thought, it would be useful to report about the two proofs.

#### 3.4.1 Core algorithm based proof

Beier and Vöcking [2004] combined the *core algorithm* by Goldberg and Marchetti-Spaccamela [1984] and the algorithm developed by Nemhauser and Ullmann [1969]. We describe these two algorithms before describing the algorithm by Beier and Vöcking [2004]. It is assumed that the weights and profits are drawn independently and uniformly at random from \([0,1]\), the value of \( n \) is assumed to be chosen according to the Poisson distribution with parameter \( N \). Further more the knapsack capacity is taken to be \( W = \beta N \), for some constant \( \beta \in (0, \frac{1}{2}) \).

**Core algorithms**

Core algorithms start by computing an optimal solution for the relaxed or fractional knapsack problem. The LP solution is obtained by starting with the empty knapsack and adding one by
one the items in order of non-increasing profit-to-weight ratio. The algorithm stops when the first item can not be accommodated completely in the empty space of the knapsack. This item is called the *break item* and the solutions corresponding to items selected so far not including the break item is called *break solution*. See figure 3.1

![Figure 3.1: Break item and Dantzig ray](image)

For a geometrical interpretation, each of the item with profit $p_i$ and weight $w_i$ can be represented by a point $(w_i, p_i)$ in the unit square on the 2-D plane. The greedy algorithm can be thought of as rotating a ray clockwise around the origin starting from the vertical (profit) axis and inserting items as an when the ray swipes the point representing the item. The rotation stops when it reaches the break item. The position of the ray defined by the breaking item is called *Dantzig ray*.

There is a strong motivation to start with the break solution. In most of the practical examples, it is observed that the optimal solution differs from the break solution by items near the Dantzig ray. The items far above the dantzig ray is always included in the optimal solution and items far below are not included. That means, we can reduce our search with items near the Dantzig ray on both sides. These items are called *core items*. However, it remains to define the term near precisely.

Before going the precise definition of nearness, let us define few terms. Assume that the items are already given in the order of non-increasing profit-to-weight ratio and $k$ is the index of the break item. The solution vector of the fractional solution will look like $\bar{a} = (1, \ldots, 1, f, 0, \ldots, 0)$ where $f \in (0, 1]$ is at the position $k$. For any feasible integer solution $a = (a_1, \ldots, a_n)$ define $ch(a) = \{i \in [n] : \bar{a}_i \neq a_i\}$ i.e. $ch(a)$ is the vector of indices of the items for which the components of $a$ and $\bar{a}$ do not agree. Let $r$ be the profit-to-weight ratio of the break item i.e. $r = \frac{p_k}{w_k}$. Define $loss(i) = |p_i - rw_i|$, i.e. $loss(i)$ is the vertical distance of the item from Dantzig
ray. Define profit of a solution $a$ as $P(a) = \sum_{i \in [n]} a_i p_i$. Given the above definitions, Goldberg and Marchetti-Spaccamela [1984] proved the following relation between the profit of LP solution and the profit of any integer solution

\[ P(\bar{a}) - P(a) = r \left( W - \sum_{i \in [n]} a_i w_i \right) + \sum_{i \in ch(a)} loss(i) \quad (3.24) \]

Let $a^*$ be the optimal integer solution. Then the integrality gap $\Gamma = P(\bar{a}) - P(a^*)$ is given by

\[ \Gamma = r \left( W - \sum_{i \in [n]} a^*_i w_i \right) + \sum_{i \in ch(a^*)} loss(i) \quad (3.25) \]

\[ \Rightarrow \sum_{i \in ch(a^*)} loss(i) \leq \Gamma \quad (3.26) \]

Since $loss(i)$ is a positive quantity, each of $loss(i)$ for the items in $ch(a^*)$ is upper bounded by $\Gamma$. Thus all items for which the integer optimal solution differs from the LP optimal solution, must be at within vertical distance of $\Gamma$ from the Dantzig ray. This gives a precise notion of nearness for defining the core items. However, the problem is that we do not know the integrality gap $\Gamma$ in advance. The difficulty is solved by the following lemma by Goldberg and Marchetti-Spaccamela [1984].

**Lemma 3.4.1** There is a constant $c_0$ such that for every $\alpha \leq \log^4 N$, $\Pr \left[ \Gamma \geq c_0 \alpha \frac{\log^2 N}{N} \right] \leq 2^{-\alpha}$.

In words, the integrality gap does not exceed $O\left(\frac{\log^2 N}{N}\right)$, with high probability.

Now we describe the algorithm by Nemhauser and Ullmann [1969].

**The Nemhauser/Ullman algorithm**

The algorithm is based on the dominating sets. Before defining dominating sets, let us have the following definition. For a subset $S \subseteq [n]$, define weight $w(S) = \sum_{i \in S} w_i$ and profit $p(S) = \sum_{i \in S} p_i$. A subset $S$ dominates another subset $T \subseteq [n]$ if $w(S) \leq w(T)$ and $p(S) \geq p(T)$. A subset which is not dominated by any other set is called a dominating set.

Now, for simplicity, let us assume that all possible subsets of the items have different profits. Then the optimal subset should be from all the possible dominating sets. Hence, if we can somehow find out the dominating sets, then we can restrict our search for finding out the optimal, within those dominating sets. Nemhauser and Ullmann [1969] introduced the following elegant algorithm that computes the sequence of dominating sets in a very efficient way.

For $i \in [n]$, let $S_i$ be the sequence of dominating subsets over the items $1, \ldots, i$. The sets in $S_i$ are assumed to be listed in increasing order of their weights. Given $S_i$, the sequence $S_{i+1}$ can be computed from $S_i$ as follows. First duplicate all subsets in $S_i$ and then add item $i+1$ to the new copy. Thus we have two ordered sequences of sets. The two sequences are merged by removing those subsets in one sequence that are dominated by some other subset in the other sequence. Geometrically this can be viewed as follows. Each of the two sequences takes the form of a non-decreasing step graph if the profits of the subsets are plotted in the Y-axis and weights in the X-axis. The resulting sequence will be the step graph generated by taking the
3.4. RANDOM KNAPSACK IN POLYNOMIAL TIME

graph which lies above the other. The result is the ordered sequence $S_{i+1}$ of dominating sets over the items $1, \ldots, i + 1$.

The solution to the knapsack problem can be obtained by computing the sequence $S_n$ and then taking the subset in $S_n$ which does not exceed the knapsack capacity and gives maximum profit. The following lemma helps in computing the running time of the Nemhauser/Ullman algorithm.

**Lemma 3.4.2** For every $i \in [n]$, let $q(i)$ denote the number of dominating sets over items $1, \ldots, i$ and assume $E[q(i+1)] \geq E[q(i)]$. The Nemhauser/Ullman algorithm solves the knapsack problem in expected time $O(\sum_{i=1}^{n-1} E[q(i)]) = O(nE[q(n)])$.

In another paper (), the authors shown that $E[q(n)] = O(n^3)$ for uniformly random instances. Hence, the expected running time of the algorithm for these instances is $O(n^4)$. This alone proves the polynomial average-case complexity of the knapsack problem. However, the authors improved the running time from $O(n^4)$ to $O(n \text{polylog} n)$ by combing it with the core algorithms. It is discussed in the next section.

**Combined algorithm**

The steps of the combined algorithm is as follows

1. Generate a static core with loss at most $d = cdN^{-1} \log^3 N$, with a suitable constant $c_d$.
2. Use Nemhauser/Ullman algorithm to generate the dominating sets over these core items.
3. Among all the dominating sets satisfying the capacity bound, the most profitable one is selected.
4. If the profit of this set differs from the profit of the optimal fractional solution by at most $d$ then return the selected dominating set. Otherwise output failure.

The correctness of the algorithm follows from the discussion on core algorithm. If the integrality gap of the output solution is at most $d$, it is guaranteed that $\Gamma \leq d$ and hence the output is correct. On the otherhand, the algorithm fails when $d < \Gamma$. Here the algorithm is conservative in the sense that the algorithm may nevertheless have found the correct solution but it is not sure about correctness.

The authors improved the algorithm, with two different lists of dominating sets, to give exact solution always which is omitted for simplicity. The authors gives the proof that the running time of the algorithm is $O(N \text{polylog} N)$. Here give a very intuitive proof. The expected number of the core items corresponds to $N$ times the area covered by the $d$-region around the Dantzig ray $\approx 2dN$. Using lemma 3.4.2, expected run time $= O(N \cdot 2dN) = O(N^2 c_d N^{-1} \log^3 N) = O(N \text{polylog} N)$.

**3.4.2 Isolating lemma based proof**

The dynamic programming algorithm used in the polynomial time approximation scheme (the lecture notes on Approximation Algorithms by David Williamson can be referred) runs in time
polynomial in \( V \), the maximum profit of items. This is polynomial time if \( V \) is restricted to be \( O(\log n) \) bits long, where \( n \) is the number of items. The key idea is that when the values of the items are generated at random, it might suffice to approximate the values to about \( 2\log n \) bits, thus making the algorithm polynomial time and not pseudo-polynomial time.

Imagine that we generate values one bit at a time. Initially, there will be many candidate subsets which will give the maximum value. As more and more bits of the values are generated, the number of subsets giving max value will dwindle down, until eventually only one subset remains. In the limit, if there is a unique item with maximum profit, the item can be identified after generating \( 2\log n \) bits. So, this will give a polynomial time algorithm. However, we need to show that there is a unique item with maximum profit.

The Isolating Lemma Mulmuley et al. [1987] helps in this case. Before using the lemma, let us have the following definition.

Let \( X \) be some set of \( n \) points and \( \mathcal{F} \) be a family of subsets of \( X \). Let us assign a weight \( w(x) \) to each point \( x \in X \) and let us define the weight of a set \( E \) to be \( w(E) = \sum_{x \in E} w(x) \). It may happen that several sets of \( \mathcal{F} \) will have the minimal weight. If it is not the case, i.e. if \( \min_{E \in \mathcal{F}} \) is achieved by a unique \( E \in \mathcal{F} \), then we say that \( w \) is isolating for \( \mathcal{F} \).

Now, according to the lemma, independent of what our family \( \mathcal{F} \) actually is, a randomly chosen \( w \) is isolating for \( \mathcal{F} \) with high probability. Since max function is negative of min, the same apply for for max function too. In our case, the profit is the weight function isolating and drawn randomly. We take the family of sets \( \mathcal{F} \) to be the sets with only one element. Hence there will be a unique maximum with high probability.

### 3.5 An experimentation on accelerating column generation

We mentioned in the section on our learning from literature survey (see 3.2) that one way to improve the performance is to accelerate the column generation process while solving the LP relaxation at any node in the branch and bound tree. For solving the LP relaxation, initially, we take a very restricted or constrained form of the master problem with only few columns enough to ensure the feasibility and gradually relaxing the formulation by adding new columns. This is continued until we get an optimal solution. Any trickier formulation to accelerate the finding of the optimal set of columns should make this relaxation process faster.

To find such a formulation that relaxes the master problem, we thought of allowing the demand of an item of smaller width be met by an item of larger width. Mathematically, suppose we need to cut \( b_1 \) number of items of width \( w_1 \) and \( b_2 \) number of items of width \( w_2 \) where \( w_1 \geq w_2 \). We can relax this requirement by saying that we require \( b_1 \) number of items of width \( w_1 \) and together \( b_1 + b_2 \) number of items of width either \( w_2 \) or \( w_1 \). This is reasonable because if in the solution the demand of \( w_2 \) is met by \( w_1 \), we can easily replace \( w_1 \) by \( w_2 \) in the corresponding cutting pattern though the cutting may not be optimal.

In the following section we show that such a relaxation in the master LP, adds the constraint in the dual that the corresponding dual variables are also in the same order as the widths of the items.
3.5. AN EXPERIMENTATION ON ACCELERATING COLUMN GENERATION

3.5.1 Formulation

The constraints in the master LP is as follows

\[ A_1 \lambda \geq b_1 \] \hspace{1cm} (3.27)
\[ A_2 \lambda \geq b_2 \] \hspace{1cm} (3.28)
\[ \ldots \] \hspace{1cm} (3.29)
\[ A_n \lambda \geq b_n \] \hspace{1cm} (3.30)

where \( A_i \) is the \( i^{th} \) row of the matrix \( A \).

If we apply the relaxation as described above, the constraints take the form

\[ A_1 \lambda' \geq b_1 \] \hspace{1cm} (3.31)
\[ A_1 + A_2 \lambda' \geq b_1 + b_2 \] \hspace{1cm} (3.32)
\[ \ldots \] \hspace{1cm} (3.33)
\[ A_1 + A_2 + \ldots + A_n \lambda' \geq b_1 + b_2 + \ldots + b_n \] \hspace{1cm} (3.34)

note that solution to the relaxed system may not be a solution to the original system and that is why new variable \( \lambda' \) is used.

It is equivalent to premultiplying both sides by a lower triangular matrix \( L \) with all non-zero elements 1 i.e.

\[
L = \begin{bmatrix}
1 & 0 & \ldots & 0 \\
1 & 1 & \ldots & 0 \\
\vdots & \vdots & \ddots & \vdots \\
1 & 1 & \ldots & 1
\end{bmatrix}
\] (3.35)

Thus, the related master LP is

\[
\begin{aligned}
\min & \quad 1^T \lambda' \\
\text{s.t.} & \quad LA\lambda' \geq Lb \\
& \quad \lambda' \geq 0
\end{aligned}
\] (3.36)\hspace{1cm} (3.37)\hspace{1cm} (3.38)

The corresponding dual is

\[
\begin{aligned}
\max & \quad b^T L^T \pi' \\
\text{s.t.} & \quad A^T L^T \pi' \leq 1 \\
& \quad \pi' \geq 0
\end{aligned}
\] (3.39)\hspace{1cm} (3.40)\hspace{1cm} (3.41)

If we substitute \( L^T \pi' \) by \( \pi \), i.e. \( \pi' = (L^T)^{-1} \pi \)

\[
\begin{aligned}
\max & \quad b^T \pi \\
\text{s.t.} & \quad A^T \pi \leq 1 \\
& \quad (L^T)^{-1} \pi \geq 0
\end{aligned}
\] (3.42)\hspace{1cm} (3.43)\hspace{1cm} (3.44)

However, \( (L^T)^{-1} \) is given by

\[
(L^T)^{-1} = \begin{bmatrix}
1 & -1 & 0 & \ldots & 0 \\
0 & 1 & -1 & \ldots & 0 \\
\vdots & \vdots & \ddots & \vdots & \vdots \\
0 & 0 & \ldots & 1 & -1 \\
0 & 0 & \ldots & 0 & 1
\end{bmatrix}
\] (3.45)
3.6. SUMMARY

The extra constraint (3.44) implies

\[ \pi_1 \geq \pi_2 \geq \cdots \geq \pi_n \]  
(3.46)

3.5.2 Proof of correctness

It needs to prove that with the relaxed primal, the solution obtained at the end of column generation process, the objective would be same as that objective that would have been obtained if the relaxation was not used. In the experimental instances we have seen the two objectives to match though we are in the process of finding a proof. However, we should not be disappointed because at the end, we could remove the relaxation and apply column generation few more steps to obtain the LP optimal. This also may not be required because often the LP is not solved till optimality in the branch-and-price process.

3.5.3 COIN-OR based implementation

For the implementation of the above scheme, we were looking for an existing open source framework so that we do not have to rewrite code afresh for many of the subtasks. We used the COIN-OR framework, \texttt{http://www.coin-or.org}, for our experimentation. The good thing about this is that it already provides a framework for branch-cut-price algorithms which will help in experimentation planned in the next stage of our project.

3.5.4 Experimental results

In this experiment, we constructed CSP example with random widths in the range 1..1,000 and roll-width 10,000 such that the widths fit exactly in two rolls. The result is shown in the table 3.1. In the table, \textit{basic} represents the case when the basic Gilmore-Gomory method of column generation is used. \textit{goodp} represents the use of relaxation as described in the above section.

We observed that the new scheme took on an average 55\% less number of iterations compared to the simple column generation process. However, we found that overall time taken is increased by 9\% on an average. Increase in time is due to the fact that the knapsack problem becomes harder to solve.

Though the test coverage is not much, from this experiment and similar experiment done in Pal [2006], it is clear that the relaxation is providing much gain in terms of number of iterations but the gain is reduced by the extra time in solving the knapsack time. This implies that we need to look into how the knapsack can be solved better.

Another interesting point is that the gain may fructify in the over all branch and price process where the LP is not often solved to optimality. This will be experimented in the next stage of our work.

3.6 Summary

We completed pending survey of literature on branch-and-price and provided the conclusion. We computed the worst case complexity of a dynamic programming solution and compared it with the worst case complexity of the standard branch-and-bound algorithm for solving such
### Table 3.1: Experimental result for the new technique of relaxing the master problem

<table>
<thead>
<tr>
<th>Test No</th>
<th>Heuristic</th>
<th>Num Iteration</th>
<th>Gain NT %</th>
<th>Solver Time</th>
<th>Gain ST %</th>
<th>Primal Time</th>
<th>Gain PT %</th>
<th>Knapsack Time</th>
<th>Gain KT %</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>basic</td>
<td>163.00</td>
<td>0.00</td>
<td>35.32</td>
<td>0.00</td>
<td>0.24</td>
<td>0.00</td>
<td>35.08</td>
<td>0.00</td>
</tr>
<tr>
<td>1</td>
<td>goodp</td>
<td>67.00</td>
<td>58.90</td>
<td>25.91</td>
<td>26.66</td>
<td>0.11</td>
<td>53.33</td>
<td>25.79</td>
<td>26.48</td>
</tr>
<tr>
<td>2</td>
<td>basic</td>
<td>160.00</td>
<td>0.00</td>
<td>47.74</td>
<td>0.00</td>
<td>0.28</td>
<td>0.00</td>
<td>47.46</td>
<td>0.00</td>
</tr>
<tr>
<td>2</td>
<td>goodp</td>
<td>65.00</td>
<td>59.38</td>
<td>36.04</td>
<td>24.52</td>
<td>0.10</td>
<td>64.79</td>
<td>35.94</td>
<td>24.28</td>
</tr>
<tr>
<td>3</td>
<td>basic</td>
<td>144.00</td>
<td>0.00</td>
<td>30.87</td>
<td>0.00</td>
<td>0.19</td>
<td>0.00</td>
<td>30.68</td>
<td>0.00</td>
</tr>
<tr>
<td>3</td>
<td>goodp</td>
<td>62.00</td>
<td>56.94</td>
<td>39.84</td>
<td>-29.07</td>
<td>0.08</td>
<td>56.25</td>
<td>39.76</td>
<td>-29.60</td>
</tr>
<tr>
<td>4</td>
<td>basic</td>
<td>119.00</td>
<td>0.00</td>
<td>39.07</td>
<td>0.00</td>
<td>0.10</td>
<td>0.00</td>
<td>38.98</td>
<td>0.00</td>
</tr>
<tr>
<td>4</td>
<td>goodp</td>
<td>61.00</td>
<td>48.74</td>
<td>36.41</td>
<td>6.82</td>
<td>0.06</td>
<td>37.50</td>
<td>36.35</td>
<td>6.74</td>
</tr>
<tr>
<td>5</td>
<td>basic</td>
<td>158.00</td>
<td>0.00</td>
<td>26.08</td>
<td>0.00</td>
<td>0.26</td>
<td>0.00</td>
<td>25.82</td>
<td>0.00</td>
</tr>
<tr>
<td>5</td>
<td>goodp</td>
<td>70.00</td>
<td>55.70</td>
<td>23.47</td>
<td>10.00</td>
<td>0.11</td>
<td>57.58</td>
<td>23.36</td>
<td>9.51</td>
</tr>
<tr>
<td>6</td>
<td>basic</td>
<td>163.00</td>
<td>0.00</td>
<td>48.52</td>
<td>0.00</td>
<td>0.29</td>
<td>0.00</td>
<td>48.22</td>
<td>0.00</td>
</tr>
<tr>
<td>6</td>
<td>goodp</td>
<td>70.00</td>
<td>57.06</td>
<td>60.81</td>
<td>-25.34</td>
<td>0.12</td>
<td>60.27</td>
<td>60.69</td>
<td>-25.86</td>
</tr>
<tr>
<td>7</td>
<td>basic</td>
<td>142.00</td>
<td>0.00</td>
<td>42.06</td>
<td>0.00</td>
<td>0.19</td>
<td>0.00</td>
<td>41.87</td>
<td>0.00</td>
</tr>
<tr>
<td>7</td>
<td>goodp</td>
<td>71.00</td>
<td>50.00</td>
<td>49.15</td>
<td>-16.85</td>
<td>0.10</td>
<td>48.93</td>
<td>49.06</td>
<td>-17.15</td>
</tr>
<tr>
<td>8</td>
<td>basic</td>
<td>133.00</td>
<td>0.00</td>
<td>24.27</td>
<td>0.00</td>
<td>0.16</td>
<td>0.00</td>
<td>24.11</td>
<td>0.00</td>
</tr>
<tr>
<td>8</td>
<td>goodp</td>
<td>64.00</td>
<td>51.88</td>
<td>44.24</td>
<td>-82.28</td>
<td>0.10</td>
<td>40.00</td>
<td>44.14</td>
<td>-83.09</td>
</tr>
<tr>
<td>9</td>
<td>basic</td>
<td>164.00</td>
<td>0.00</td>
<td>27.21</td>
<td>0.00</td>
<td>0.30</td>
<td>0.00</td>
<td>26.91</td>
<td>0.00</td>
</tr>
<tr>
<td>9</td>
<td>goodp</td>
<td>69.00</td>
<td>57.93</td>
<td>23.13</td>
<td>15.02</td>
<td>0.12</td>
<td>60.00</td>
<td>23.01</td>
<td>14.52</td>
</tr>
<tr>
<td>10</td>
<td>basic</td>
<td>161.00</td>
<td>0.00</td>
<td>43.71</td>
<td>0.00</td>
<td>0.24</td>
<td>0.00</td>
<td>43.47</td>
<td>0.00</td>
</tr>
<tr>
<td>10</td>
<td>goodp</td>
<td>63.00</td>
<td>60.87</td>
<td>49.88</td>
<td>-14.11</td>
<td>0.10</td>
<td>57.38</td>
<td>49.78</td>
<td>-14.51</td>
</tr>
</tbody>
</table>

Average: Gain NT % = 55.74, Gain ST % = -8.46, Gain PT % = 53.6, Gain KT % = -8.87
3.6. SUMMARY

problems. We mentioned the literature we found that solves the knapsack problem in polynomial time on an average. At the end, we described the our experimentation on accelerating the column generation process.
Chapter 4

Work planned for third stage

In this chapter we give an outline of the work that we want to undertake in the next stage of our project. From the experimentation done stage 2, it came out that we need to focus on obtaining the integer solution.

4.1 Comparative study of the existing algorithms

Before staring our own branching algorithm, we want to verify the existing ideas on our setup. There are few confusing comments made in some of the papers we studied. For example, in the first paper Vance et al. [1994] mentioned that the conventional branching is not suitable because it does not divide the problem space into equal sizes. That is the reason they gave for using the proposition 2.2.1 for branching. However, in the newer paper (Vance [1998]) they used a branching scheme that is equivalent to branching on a single variable. We need to know the reason behind that.

Also, we need to make a comparative study of the algorithms that we surveyed. It seems to be a difficult task because

- All of them did not cover the same set of experimental instances.
- They undertook experiments on different types of machines.

It is not possible the know exactly which algorithm is better on what set of test instances. Now, since our COIN-OR based framework can accommodate all these algorithms, we can have a fair comparison among them.

4.2 Work on getting the integer solution

As we mentioned in section 3.5.4 it is clear that working only on accelerating the column generation for solving the LP relaxation will not help. We need to see how this fructify in the overall branch-and-bound process. For that we have to start working on getting the integer solution.

We expect that the comparative study that we are going to undertake will guide us in the direction to follow. From our current knowledge, the branching scheme of Vanderbeck [1999]
is the best to follow. Because, this is the only method which handle the general cutting stock problem.

4.3 Summary

We outlined the work for the next stage of our work. The idea is to do the comparative study of the existing algorithms on our COIN-OR based setup and then pursue experimentation on the branching scheme described in Vanderbeck [1999].
Chapter 5

Conclusions

Using branch-and-price is a success story in solving large scale integer programming. The advantage of keeping only few columns at any time, makes many real life integer optimization problems thought of being solved. When used with extensive formulation having stronger linear programming relaxation, the method gives quicker result. The technique of combining column generation with the branch-and-bound technique provides the possibility of solving the large problems exactly. We hope that the overall solution can be made efficient by using efficient method of branching and efficient way of solving the subproblem even in the presence of extra constraints due to branching.
References


