New solving methods for constrained optimization problems

by

Amine Lamine

PhD Thesis
In Computer System Engineering

Supervised by
Prof. Dr. Habib Chabchoub
Prof. Dr. Brahim Hnich
Dr. Mahdi Khemakhem

Academic year
2016-2017
Acknowledgements

First and foremost, I would like to express my best thanks to everyone who contributed to my personal and professional development before and during my whole Ph.D. studies.

I would, first of all, like to express the sincerest thanks to my supervisor Professor Habib Chabchoub. I’m most of obliged to him for giving me an opportunity to work within his team. It was a great experience for me. Thank you for your encouragement, support, and guidance.

I would also like to thank my co-supervisor Doctor Mahdi Khemakhem, for always being very helpful since my B.A.; especially for guiding my research to this topic, providing important information from OR and CP fields, helping me write my thesis and publish my papers and reviewing them several times. I would have never been able to complete this thesis without his patient and guidance. I am really grateful to him and I would like to thank him for all precious time that he has devoted to this thesis and all supports during stressful moments of discouragement.

I also wish to express my gratitude to my external collaborator Professor Brahim Hnich, for always being very helpful during my stay at the Izmir University of Economics. From my point of view, he did one of the most influential works in constraint programming. Thank you Brahim for all your helpful advice during my research visit. Thanks for many useful discussions, explaining many matters to me, support, significant help in the methodology of writing a paper, and last and not least your friendship.

Eventually, I would like to thanks my whole family. I am grateful to my parents for encouraging me in this work, helping me continue with this lengthy quest. My sincere thanks to my brothers and my sister for all the motivation, kindness, love and affection throughout my studies. I extend my great thanks to my wife and
my in-laws. She supports me everyday. She’s always there to help me to find a good balance between research, work and family.
# Contents

Acknowledgements i  
List of Figures vi  
List of Tables ix  

1 Introduction 1  
1.1 Knapsack Problems 1  
1.2 Constraint programming 2  
1.3 Contributions 4  
1.4 Structure of the thesis 4  

2 Preliminaries: an overview of constraint programming 7  
2.1 Introduction 7  
2.2 Variables, Domains and Constraints 8  
2.3 Propagation and Search 9  
2.3.1 Propagation 9  
2.3.2 Search 10  
2.4 Global Constraint 12  
2.5 Optimization 14  

3 The multiple demand multidimensional multiple choice knapsack problem: definition and relationship problems 17  
3.1 Introduction 17  
3.2 Preliminaries 18  
3.2.1 Integer linear program 18  
3.2.2 Knapsack problem constraints 19  
3.2.3 Reduction, generalization and problems transformation 20  
3.3 The knapsack problem family involving the notion of dimensions, demands and sets 21  
3.3.1 The knapsack problem 21  
3.3.2 The multidimensional knapsack problem 22  
3.3.3 The multiple demand multidimensional knapsack problem 24  
3.3.4 The multiple choice knapsack problem 25
### 3.3.5 The multidimensional multiple choice knapsack problem

26

### 3.3.6 The multidimensional knapsack problems with generalized upper bound constraints

28

### 3.3.7 The multiple demand multidimensional multiple choice knapsack problem

29

### 3.3.8 Relation schema between problems

31

### 3.4 Transformations between Integer Linear Programs

32

#### 3.4.1 Transformation of the GUBMKP into the MMKP

33

#### 3.4.2 Transformation of the MKP into the MMKP

34

#### 3.4.3 Transformation of the GUBMKP into the MKP

35

#### 3.4.4 Transformation of the MMKP into the MDMKP

37

#### 3.4.5 Transformation of the MCKP into the GUBMKP

37

#### 3.4.6 Algorithms of MDMMKP are able to solve the other problems

39

### 3.5 Experimental results

40

#### 3.5.1 Instances details

41

#### 3.5.2 Evaluation of the transformation

44

### 3.6 Conclusion

46

### 4 The multiple choice multidimensional knapsack constraint

48

#### 4.1 Introduction

48

#### 4.2 Constraint programming preliminaries

49

##### 4.2.1 Constraint programming

50

##### 4.2.2 \( \sum \) and \( \text{implies} \) constraints

51

#### 4.3 The multiple choice multidimensional knapsack constraint

51

##### 4.3.1 Fundamental properties

52

#### 4.4 Filtering algorithm for \( mcmdk \) constraint

55

##### 4.4.1 A worked example

57

#### 4.5 Experiments

58

#### 4.6 Conclusion

60

### 5 Solving Constrained Optimization Problems By Solution-based Decomposition Search

61

#### 5.1 Introduction

61

#### 5.2 Formal background

62

#### 5.3 The basic Solve and Decompose algorithm

63

##### 5.3.1 The decomposition method

64

##### 5.3.2 Identification of promising subproblems

64

##### 5.3.3 Strengthening the cost-based filtering

65

##### 5.3.4 Decomposition-based search

66

##### 5.3.5 Example

69

#### 5.4 Improving Solve and Decompose

70

#### 5.5 Computational results

71

##### 5.5.1 Benchmark problems

71

###### 5.5.1.1 MMKP

71

###### 5.5.1.2 SMSDP

72
### Contents

5.5.2 Settings ............................................. 72  
5.5.3 Results ............................................. 73  
5.6 Related work ........................................... 76  
5.7 Conclusion ............................................ 78  

6 Conclusions and Future Work 89  
6.1 Summary and conclusions ............................. 89  
6.2 Discussion and Future work .......................... 90  

Bibliography 92
List of Algorithms

- Function constraintSearch\((D)\) ............................................. 11
- Procedure initialize\((Q,m,n:Integer)\) ................................. 55

1 Filtering algorithm ....................................................... 56

2 Basic \(S&D\) ............................................................... 81

3 \(solveAndDecompose(pb,f,depth,levelID)\) ........................ 81

4 \(improvedSolveAndDecompose(pb,f,depth,levelID)\) .............. 81
List of Figures

2.1 An example of constraint satisfaction problem. .................. 9
2.2 A general form of an integer linear program .................... 16

3.1 A general form of an integer linear program .................... 18
3.2 An integer linear program of the knapsack problem ............. 22
3.3 An instance example of the knapsack problem .................. 22
3.4 An integer linear program of the multidimensional knapsack problem 23
3.5 An instance example of the multidimensional knapsack problem .. 23
3.6 An integer linear program of the multiple demand multidimensional knapsack problem .................. 24
3.7 An instance example of the multiple demand multidimensional knapsack problem .................. 25
3.8 An integer linear program of the multiple choice knapsack problem 26
3.9 An instance example of the multiple choice knapsack problem .. 27
3.10 An integer linear program of the multidimensional multiple choice knapsack problem .................. 27
3.11 An instance example of the multidimensional multiple choice knapsack problem .................. 28
3.12 An integer linear program of the multidimensional knapsack problems with generalized upper bound constraints .......... 29
3.13 An instance example of the multidimensional knapsack problems with generalized upper bound constraints .......... 30
3.14 An integer linear program of the multidimensional multiple demand multiple choice knapsack problem .................. 30
3.15 An instance example of the multidimensional multiple demand multiple choice knapsack problem .................. 31
3.16 Relation between knapsack problems ............................ 32
3.17 Transformation between ILPs of KPs ............................ 33
3.18 An integer linear program of GUBMKP based on the MMKP formulation .................................................. 33
3.19 An instance example of the GUBMKP based on the MMKP formulation .................................................. 34
3.20 An integer linear program of MKP based on the MMKP formulation .................................................. 35
3.21 An instance example of the MKP based on the MMKP formulation .................................................. 36
3.22 An integer linear program of MKP based on the MMKP formulation .................................................. 37
3.23 An instance example of the GUBMKP based on the MKP formulation .................................................. 38
<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.24</td>
<td>A second integer linear program of MMKP</td>
<td>39</td>
</tr>
<tr>
<td>3.25</td>
<td>An integer linear program of MMKP based on the MDMKP formulation</td>
<td>39</td>
</tr>
<tr>
<td>3.26</td>
<td>An instance example of the MMKP based on the MDMKP formulation</td>
<td>40</td>
</tr>
<tr>
<td>3.27</td>
<td>An integer linear program of MCKP based on the GUBMKP formulation</td>
<td>41</td>
</tr>
<tr>
<td>3.28</td>
<td>An instance example of the MCKP based on the GUBMKP formulation</td>
<td>42</td>
</tr>
<tr>
<td>4.1</td>
<td>An instance example of the multiple choice multidimensional knapsack constraint</td>
<td>53</td>
</tr>
<tr>
<td>5.1</td>
<td>A Trace of the basic S&amp;D on an example.</td>
<td>82</td>
</tr>
<tr>
<td>5.2</td>
<td>Multidimensional multiple choice knapsack problem formulation.</td>
<td>82</td>
</tr>
<tr>
<td>5.3</td>
<td>Steel Mill Slab Design Problem formulation.</td>
<td>82</td>
</tr>
<tr>
<td>5.4</td>
<td>Time consumed by S&amp;D in each level for MMKP instance inst17</td>
<td>83</td>
</tr>
</tbody>
</table>
List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>Knapsack problems according to the type of constraints</td>
<td>20</td>
</tr>
<tr>
<td>3.2</td>
<td>Notation of knapsack terms</td>
<td>20</td>
</tr>
<tr>
<td>3.3</td>
<td>Test MKP instances details</td>
<td>42</td>
</tr>
<tr>
<td>3.4</td>
<td>Test MDMKP instances details</td>
<td>43</td>
</tr>
<tr>
<td>3.5</td>
<td>Test MMKP instances details</td>
<td>43</td>
</tr>
<tr>
<td>3.6</td>
<td>Test GUBMKP problem details</td>
<td>44</td>
</tr>
<tr>
<td>3.7</td>
<td>Test MCKP instances details</td>
<td>44</td>
</tr>
<tr>
<td>3.8</td>
<td>Performances comparison of the transformation between different ILPs</td>
<td>45</td>
</tr>
<tr>
<td>4.1</td>
<td>A worked example</td>
<td>57</td>
</tr>
<tr>
<td>4.2</td>
<td>Comparison on instances with $n = 15$</td>
<td>59</td>
</tr>
<tr>
<td>4.3</td>
<td>Comparison on instances with $m = 15$</td>
<td>60</td>
</tr>
<tr>
<td>5.1</td>
<td>The runtime in second for MMKP instances using basic $S&amp;D$ and improved $S&amp;D$</td>
<td>84</td>
</tr>
<tr>
<td>5.2</td>
<td>The runtime in second for MMKP instances using basic $S&amp;D$ and improved $S&amp;D$</td>
<td>85</td>
</tr>
<tr>
<td>5.3</td>
<td>The runtime in second for SMSDP instances using basic $S&amp;D$ and improved $S&amp;D$</td>
<td>86</td>
</tr>
<tr>
<td>5.4</td>
<td>The runtime of $S&amp;D$ in second for MMKP instances using different $depth$</td>
<td>87</td>
</tr>
<tr>
<td>5.5</td>
<td>The runtime of $S&amp;D$ in second for SMSDP instances using different $depth$</td>
<td>88</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

The objective of this thesis is (i) to study some relationships between knapsack problems, which are classical constrained optimization problems (COPs), (ii) provide a solution using constraint programming for a knapsack problem generalization and, (iii) present a new strategy for solving COPs. In this chapter, we introduce knapsack problems and constraint programming in turn, summarize the main contributions of the thesis and provide the outline of the document.

1.1 Knapsack Problems

Knapsack problems are among the most extensively studied $NP$-hard combinatorial optimization problems. The knapsack problems have been used to model a wide range real-world applications ranging from cargo loading, project selection, cutting stock, capital budgeting to resource allocation problems and more.

The classical knapsack problem considers a set of items, each having an associated profit and a weight. The problem is to choose a subset of the given items such that the corresponding total profit is maximized while the total weight satisfies a specified resource capacity.
In industry, a large number of industrial applications find the need for satisfying additional specific constraints such as resource capacity and item weight are multidimensional besides, selecting items with different weight requirements from different sets. These necessities lead to a set of variants and extensions of knapsack problems. Among these extensions, we cite the multidimensional knapsack problem, the multiple demand multidimensional knapsack problem and the multiple choice multidimensional knapsack problem. These involve the notion of sets and dimensions.

1.2 Constraint programming

Constraint programming (CP) is a powerful programming paradigm for solving large-scale combinatorial problems, wherein relations between variables are stated by means of constraints.

The CP paradigm is by nature declarative. It declares what constraints need to be satisfied. This phase is called ”constraint specification” and is separate from the search phase also known as ”constraint solver”, which determine how the solutions are built. CP has proven efficient and effective in a wide range of application areas ranging from timetabling, scheduling, resource allocation, program verification to vehicle routing and beyond.

The constraint specification, also called model, is typically specified in a constraint modeling language. There are many well-known languages, for example OPL Van Hentenryck (1999), Choco Fages et al. (2013), MiniZinc Nethercote et al. (2007) and Essence Frisch et al. (2008), which offer powerful environments for constraint specification.

Each modeling language contains numerous variables such as integer including binary, set, graph and real variables. It contains a large number of predefined constraints as well. Typical constraints include arithmetic constraints, sequence
constraints, scheduling constraints, element constraints and compatibility con-
straints. Modeling languages also accept some powerful combinatorial constraints
known as specialized constraints such as allDifferent and lex constraints.

The merit of the most modern modeling languages is offering high level abstrac-
tions in order to be expressive. Moreover, they allow the user to model specific
constraints by decomposing them into other constraints.

A constraint solver will use the set of constraints to reduce the domains of the
variables and therefore reduce the size of the search space.

The key principle for reducing the search space is the constraint propagation, or
propagation in short. The propagation task is to remove subparts of the search
space that cannot belong to any feasible solution, by reasoning on the individual
constraints separately. Each constraint in the modeling language has its corre-
sponding filtering algorithm, or propagator, in the solver. The filtering algorithm
associated to a constraint asserts that the latter must be satisfied and preemp-
tively removes values that would violate it. In simpler terms, a filtering algorithm
takes domains as input and produces smaller domains as output by removing val-
ues from domains that do not belong to a solution. The principle of constraint
programming consists in fundamental separation of concerns regarding modeling
and solving. A model can be changed, without the need to change the underly-
ing solver and vice versa. Consequently, constraint programming has two main
advantages: (i) re-usability: constraints are building blocks that can be used in
various problems and applications, and (i) flexibility: we can be design very effi-
cient filtering algorithms, from operations research or graph theory, both of which
can be integrated into the constraints to solve large sub-problems efficiently.

These re-usability and flexibility make constraint programming a powerful paradigm
for modeling and solving various combinatorial problems.
1.3 Contributions

The main contributions of this thesis are the following:

- **Contribution 1**: We present a generalization of a set of knapsack problems involving the notion of dimensions, demands and multiple choice constraints and we define a set of transformations between the different integer linear programs of the studied problems. Using these transformations, we show that any algorithm able to solve the generalized problem can definitely solve its related problems.

- **Contribution 2**: We study some specific proprieties of the multiple choice multidimensional knapsack constraint and we propose a new combined constraint ”Global constraint” for which we give a filtering algorithm.

- **Contribution 3**: We design and develop new strategy for solving constraint optimization problems called solve and decompose. The proposed algorithm can be viewed as a systematic iterative depth-first strategy that is based on problem decomposition.

1.4 Structure of the thesis

This thesis consists of four chapters such that chapter 3, 4 and 5 are self-contained (including the defined terms for that chapter). Each one has its own purpose and it can be read by itself.

**Chapter 1**: Preliminaries: an overview of constraint programming 2.2

In this chapter we provide a brief overview of Constraint Programming.

**Chapter 2**: The multiple demand multidimensional multiple choice knapsack problem: definition and relationship problems 3
In this chapter, we studied a set of knapsack problems involving the notion of dimensions, demands and multiple choice constraints. Specifically, we presented a new problem called the multiple demand multidimensional multiple choice knapsack problem and we showed it as a generalization of other related problems. Moreover, we presented a set of transformations between the different integer linear programs of the studied problems. Using these transformations, we showed that any algorithm able to solve the generalized problem can definitely solve its related problems. Then, we tested the new integer linear programs on different sets of benchmarks using the commercial software Cplex 9.0. Computational results highlighted the ability of the generated formulations to produce a reasonable CPU time value compared with the original ones.

The chapter was previously published as:


A. Lamine and M. Khemakhem and H. Chabchoub The menu planning problem: a formal study and a practical study EURO XXIV, Lisbon.

**Chapter 3:** The multiple choice multidimensional knapsack constraint

In this chapter, we introduce a new weighted constraint: the multiple choice and the multidimensional knapsack constraints (or mcmdk constraint for short). Given a resource with dimensions each one has a limited capacity and items divided on disjoint sets where items have a weight in each dimension resource, this constraint selects exactly one item from each set so that their overall weight does not exceed any resource capacity. We show how the global mcmdk constraint can be modelled by some conjunctions of elementary constraints. We propose a filtering algorithm for propagating this global constraint. Our experimental results show that propagating
the \textit{mcmdk} constraint via the proposed filtering algorithm is effective and efficient.

\textbf{Chapter 4: Solving Constrained Optimization Problems By Solution-based Decomposition Search 5}

In this chapter we present a new strategy for solving COPs called solve and decompose (or \textit{S\&D} for short). The proposed strategy is a systematic iterative depth-first strategy that is based on problem decomposition. \textit{S\&D} uses a solution of the COP, found by any exact method, to further decompose the original problem into a bounded number of subproblems which are considerably smaller in size. It also uses the value of the feasible solution as a bound that we add to the created subproblems in order to strengthen the cost-based filtering. Furthermore, the feasible solution is exploited in order to create subproblems that have more promise in finding better solutions which are explored in a depth-first manner. The whole process is repeated until we reach a specified depth where we do not decompose the subproblems anymore but we solve them to optimality using any exact method like Branch and Bound. Our initial results on two benchmark problems show that \textit{S\&D} may reach improvements of up to three orders of magnitude in terms of runtime when compared to Branch and Bound.

The chapter consists of the research previously published in the following paper:


\textbf{Chapter 5: Conclusions and Future Work}

In this concluding chapter, the thesis is summarized and opportunities for future work are discussed.
Chapter 2

Preliminaries: an overview of constraint programming

2.1 Introduction

Constraint programming is a powerful programming paradigm for solving combinatorial problems, where relations between variables can be stated in the form of constraints. The core of constraint programming can be described by the following equation

\[ \text{Constraint Programming} = \text{Model} + \text{Propagation} + \text{Search} \]

Constraint programming is a declarative paradigm through which the user states the problem in terms of its constraints, and the solver is responsible for finding solutions. Stating the problem means choosing the variables, their initial set of possible values, and the constraints that a solution should respect.

In this chapter, we introduce the main notions of the constraint programming paradigm. Many of these notions are used in the thesis, and we believe that this short introduction should cover the essential ones. For more details we address the reader to the classical constraint programming books (Rossi et al., 2006a, Apt, 2003, Tsang, 1993)
2.2 Variables, Domains and Constraints

Let $x$ be a variable. The *domain* of $x$ is the set of values that can be assigned to $x$ and is denoted by $D(x)$. Let $X = x_1, x_2, \ldots, x_n$ be a sequence of variables. We denote $D(X) = \bigcup_{1 \leq i \leq n} D(x_i)$. In this thesis we only consider variables with *finite* domains.

A *constraint* $C$ on $X$ is defined as a subset of the cartesian product of the domains of the variables in $X$, i.e. $C \subseteq D(x_1) \times D(x_2) \times \ldots, \times D(x_n)$. A tuple $(d_1, \ldots, d_n) \in C$ is called a solution to $C$. We also say that the tuple satisfies $C$. A value $d \in D(x_i)$ for some $i = 1, \ldots, n$ is inconsistent with respect to $C$ if it does not belong to a tuple of $C$, otherwise it is consistent. $C$ is inconsistent if it does not contain a solution. Otherwise, $C$ is called consistent. When $n = 1$ the constraint $C$ is called an *unary constraint*, for $n = 2$ $C$ is called *binary constraint*. If $C$ is defined on more than two variables $n > 2$, $C$ is called *n-ary constraint* or *global constraint*.

A *Constraint Satisfaction Problem* (CSP) is defined by a triplet $(X, D, C)$ where $X$ is a set of variables, $D(X)$ is a set of domain values associated with $X$ and $C$ is a set of constraints.

A solution to a CSP is an assignment of variables to values in their respective domains so that all of the constraints are satisfied. A *consistent* CSP is a CSP for which a solution exists, otherwise it is inconsistent. A failed CSP is a CSP with an empty domain or with only singleton domains that together are not a solution to the CSP. A solved CSP is a CSP with only singleton domains that together are a solution to the CSP.

**Example:** Let $x_1, x_2, x_3$ be variables with respective domains $D_1 = \{1, 2\}$, $D_2 = \{1, 2\}$, $D_3 = \{1, 2, 3, 4, 5\}$. On these variables we impose the following constraints: $x_1 \neq x_2$, $x_2 \neq 2$, $x_1 + x_2 + x_3 \geq 6$. We denote the resulting CSP as

A solution to this CSP is $x_1 = 2$, $x_2 = 1$ and $x_3 = 4$. 
Constraints:
\[
\begin{align*}
  x_1 &\neq x_2 \\
x_2 &\neq 2 \\
x_1 + x_2 + x_3 &\geq 6
\end{align*}
\]

Decision variables and domains:
\[
\begin{align*}
x_1 &\in \{1,2\} \\
x_2 &\in \{1,2\} \\
x_3 &\in \{1,2,3,4,5\}
\end{align*}
\]

Figure 2.1: An example of constraint satisfaction problem.

2.3 Propagation and Search

The goal of constraint programming is to find one solution or all solutions to a given CSP. The solution process of constraint programming interleaves constraint propagation, or propagation in short, and search. We review each of those principles in turn.

2.3.1 Propagation

Constraint propagation, a major instrument of the efficiency of CP solvers, is a process that removes a subset or all the inconsistent values from the domains, by reasoning on the individual constraints. This process may significantly reduce large parts of the search space, and is vital to tackle combinatorially challenging problems.

Let \( C \) be a constraint on the variables \( x_1, \ldots, x_n \) with respective domains \( D_1, \ldots, D_n \). A propagation algorithm for \( C \) removes values from \( D_1, \ldots, D_n \) that do not participate in a solution to \( C \). A propagation algorithm does not have to remove all such values, as this may lead to an exponential running time due to the nature of some constraints.
One of the most interesting properties of a propagation algorithm is *arc consistency*. We say that a propagation algorithm associated with a constraint establishes arc consistency if it removes all the values of the domains involved in the constraint that are not consistent with the constraint.

We consider the CSP $P = (X, D, C)$. $P$ can be transformed into a smaller CSP $P'$ by repeatedly applying the propagation algorithm for all constraints in $C$ until there is no more domain reduction. This process is called constraint propagation. Constraint propagation is usually applied each time a domain has been changed, which happens very often during the solution process. Consequently, the propagation algorithms applied to make a CSP *locally consistent* should be as efficient as possible. The efficiency of constraint propagation is influenced by the order in which the propagation algorithms are applied, and by the efficiency of the propagation algorithms themselves.

The purpose of the constraint propagation algorithms is to achieve some form of local consistency. In general, these algorithms are not sufficient for finding a solution to a given CSP. Therefore, propagation is typically embedded into a search algorithm.

### 2.3.2 Search

The solution process of constraint programming uses a search tree, which is a particular rooted tree. The vertices of search trees are often referred to as nodes. The arcs of search trees are often referred to as branches. Further, if $(u, v)$ is an arc of a search tree, we say that $v$ is a direct descendant of $u$.

**Definition: Search tree** *Apt (2003)* Let $P$ be a CSP with a sequence of variables $X$. A *search tree* for $P$ is a (finite) tree such that:

- its nodes are CSPs,
- its root is $P$, 


begin
    $D \leftarrow \text{propagate}(D)$;
    if $D$ is a false domain then
        return;
    end
    if $\exists x \in X : |D(x)| > 1$ then
        $x \leftarrow \text{chooseVariable}(X)$;
        $D_s \leftarrow \text{splitDomain}(D(x))$;
        constraintSearch($D \cup \{x \mapsto D_s\}$);
    else
        Output solution;
    end
end

Function constraintSearch($D$)

– if $P_1, \ldots, P_m$ where $m > 0$ are all direct descendants of $P_0$, then the union of the solution sets $P_1, \ldots, P_m$ is equal to the solution set of $P_0$ for every node $P_0$.

A node $P$ of a search tree is at depth $d$ if the length of the path from the root to $P$ is $d$.

The above definition of the search tree is a very general notion. In constraint programming, a search tree is dynamically built by splitting a CSP into smaller CSPs, until a failed or a solved CSP is reached. The root node $P$ consists of the initial domain $D$ containing all the possible values of each variable. Solutions are found in the leaves of the search tree, where every variable $x_i$ has only one value in its domain $D(x)$.

Many search strategies have been proposed in the literature, the most widely used which performs CP solvers is the depth-first search, as given in Algorithm constraintSearch Schulte and Stuckey (2008). At each node in the search tree we apply constraint propagation to the corresponding CSP (line 2). As a result, we may detect that the CSP is inconsistent, or we may reduce some domains of the CSP. The search backtracks when a violation of a constraint is found (line 3). Otherwise, in each node of the search tree the algorithm branches by splitting the
domain of a variable (line 8). The search is further optimized by carefully choosing
the variable that is fixed next (line 7).

In splitting the domain of a variable, we first select a variable and then decide
how to split its domain. This process is guided by variable ordering heuristics
\texttt{chooseVariable} (line 7) and value ordering heuristics \texttt{splitDomain} (line 8). These
heuristics impose an ordering on the variables and values, respectively. The or-
dering imposed by these heuristics has a great impact on the search process Rossi
et al. (2006b).

\subsection{2.4 Global Constraint}

A good way to strengthen constraint propagation is to use global constraints van
Hoeve and Katriel (2006). Global constraints play an important role when finding
solutions because they provide a better view of the problem structure and make
the process more efficient.

There are several definitions of global constraint. A classical definition is a con-
straint that captures a relation between a non-fixed number of variables. In gen-
eral, a global constraint represents the conjunction of several constraints instead
of several simple (elementary) constraints. The idea of a global constraint is that,
by reasoning more globally, it is possible to design filtering algorithms that either
are able to remove more inconsistencies than elementary constraints, or it can
do it more efficiently. Still filtering algorithm for a global constraints must keep
reasonable time and space complexity.

Global constraints have three main advantagesRgin (2003):

\begin{itemize}
\item Expressiveness: it is more convenient to define one constraint corresponding
to a set of constraints than to define independently each constraint of this set.
\end{itemize}
• Since a global constraint corresponds to a set of constraints it is possible to deduce some information from the simultaneous presence of constraints.

• Powerful filtering algorithms can be designed because the set of constraints can be taken into account as a whole. Specific filtering algorithms make it possible to use operations research techniques or graph theory.

One of the best known global constraints is the \textit{alldiff} constraint, especially because the filtering algorithm associated with this constraint is able to establish arc consistency in a very efficient way.

In the literature, several other global constraints were proposed together with filtering algorithms. Rgin (2010) identify five different categories of global constraints: classical constraints, weighted constraints, soft constraints, constraints on meta-variables and open constraints. Below, we will present a constraint, ”\textit{Two sided knapsack constraint}\textquotedblright, belonging to the weighted constraints. This category contains constraints which are associated with some costs, like the knapsack constraint Fahle and Sellmann (2002), bin-packing constraint Schaus (2009). Usually, a summation is implied and there is a limit on it.

\textbf{Two sided knapsack constraint}

Trick Trick (2003a) proposed a variation of \textit{subset-sum constraint}, special case of a \textit{knapsack constraint} (called Knapsack Constraint). In fact, the subset sum problem takes as input a set $X = \{x_1, \ldots, x_n\}$ of $n$ integers and another integer $K$. The problem is to check if there exists a non-empty subset of $X$ whose elements sum to $K$. For example, given the set $X = \{3, 4, 7, 9, 11, 14\}$ and $K = 16$, the answer is \textit{yes} because the subset 3, 4, 9 sums to 16.

Trick Trick (2003a) proposed to consider the following variations: given a set of 0-1 variables $X = \{x_1, \ldots, x_n\}$ where each variable $x_i$ is associated with a coefficient $w_i$, a lower bound $L$ and an upper bound $U$, find an assignment of variables such that $L \leq \sum_{i=1}^{n} w_i x_i \leq U$. We represent $L$ and $U$ by a variable $z$, such that
Chapter 1. Preliminaries: an overview of constraint programming

\[ D(z) = [L, U]. \] For reasons of clarity, we will use the name \textit{two sided knapsack constraint} for the Trick knapsack constraint.

Then, the two sided knapsack constraint can be defined as:

\[
\text{two sided knapsack constraint}(x_1, \ldots, x_n, z, w) = \left\{ (d_1, \ldots, d_n, d) | \forall i \ d_i \in D(x_i), d \in D(z), d \leq \sum_{i=1}^{n} w_i d_i \right\} \cap \left\{ (d_1, \ldots, d_n, d) | \forall i \ d_i \in D(x_i), d \in D(z), \sum_{i=1}^{n} w_i d_i \leq d \right\},
\]

which corresponds to \( \min D(z) \leq \sum_{i=1}^{n} w_i x_i \leq \max D(z) \).

Trick proposed an approach derived from some dynamic programming method designed for pure KPs. The proposed algorithm is a pseudo-polynomial algorithm establishing arc consistency whose time complexity is in \( O(nU^2) \).

2.5 Optimization

Constraint programming deals with CSPs whose goal is to find a solution or all solutions to a given CSP. Often we want to find a solution to a CSP that is optimal with respect to certain criteria.

A constraint optimization problem (\textit{COP}), also called constraint satisfaction optimization problems (\textit{CSOP})\cite{Tsang1993}, is a CSP \( P \) defined on the variables \( x_1, \ldots, x_n \) and augmented with an objective function \( f : D(x_1) \times \cdots \times D(x_n) \to Q \) that maps the variables \( x_1, \ldots, x_n \) to an evaluation score.

An \textit{optimal solution} to a minimization (maximization) COP is a solution \( s \) to \( P \) that minimizes (maximizes) the value of \( f(s) \). The objective function value is often represented by a variable \( z \), together with the constraint \textit{maximize} \( z \) or \textit{minimize} \( z \) for a maximization or a minimization problem, respectively called \textit{objective constraint}. Note that, it is important that constraint propagation techniques be applied to the objective constraint.
To solve COPs, the common approach is to find an optimal solution by solving a sequence of CSPs. Several variations have been proposed and evaluated in the literature. The most widely used technique for solving COP is Branch and Bound B&B algorithm.

In CP B&B works as follows:

Initially, a backtracking search is used to find a feasible solution. A constraint is then added to the CSP which excludes solutions that are not better than this solution. A new solution is then found for the augmented CSP. This process is repeated until the resulting CSP is unsatisfiable, in which case the last solution found has been proven optimal.

The above algorithm constraintSearch can be moved to a B&B. In fact, a constraint is added on the evaluation score, this constraint is updated each time a better solution than the currently best known one is found (line 12), and hence avoids searching for solutions that are worse than that the latter.

Constraint programming is in many ways comparable with the more known field of integer linear programming. On the one hand, both constraint programming and integer linear programming offer a way of describing a problem. Moreover, they provide solution techniques to find a solution for this problem. The differences between them are exactly on these two subjects: the expressiveness of the language and the underlying solution techniques.

Integer linear programming is a general approach to formulating and solving constraint optimization problems with integer variables and linear constraints (inequalities or equalities). The aim is to optimize a linear cost function. In Integer linear programming, a model is called a program. The general form of an integer linear program (ILP) is (see Figure 2.2)

\[
\begin{align*}
& \min \text{ } c^T x \\
& \text{subject to } \begin{cases}
Ax \leq b \\
A^*x \geq b^* \\
X \\
x \in \mathbb{Z}^n
\end{cases}
\end{align*}
\]

where \( c \) is a \( n \)-vector, \( A \) is an \( m \times n \) matrix and \( b \) is an \( m \)-vector. \( x \) is the decision variables required to be integer valued. \( X \) is bounding-box-type restrictions on the variable. We refer to Chen et al. (2011) for more information about ILP fundamentals.
There are different approaches for solving integer programming problems, most of them are based on the relaxation. Relaxation means replacing the solution space of a given problem with a larger, but more easily searchable, one. Relaxations provide information to guide (through relaxed solutions) and accelerate (through bounds) the search phase. Linear relaxations are the key ingredient of Branch and Bound algorithm for ILPs. The existence of a good and fast relaxation is one of the biggest advantages of the ILP paradigm over CP.
Chapter 3

The multiple demand multidimensional multiple choice knapsack problem: definition and relationship problems

3.1 Introduction

Extensions of knapsack problem (Kellerer et al., 2004, Martello and Toth, 1990) play a significant role in the study of discrete mathematics. These extensions concern many practical problems in the real life such as the service level agreement, the model of allocation resources, computer systems design, project selection, cutting stock and cargo-loading.

In the literature, large specific algorithms are developed to solve extensions of the knapsack problem. Although there is a relation between many knapsack problems, many existing algorithms designed to solve specific problems must be re-adapted to solve other extensions. To be appropriate to the problem, the redevelopment of these algorithms is very difficult in general and can lose its specifications.
In this context, we define a new problem called the multiple demand multidimensional multiple choice knapsack problem (MDMMKP) which is considered as a generalization of its related problems studied in this chapter. Using a set of transformations, we show that any algorithm able to solve the generalized problem can definitely solve its related problems. Therefore the redevelopment of algorithms in this case is not considered.

The rest of this chapter is organized as follows. Section 3.2 gives the necessary formal preliminaries. In section 3.3, definitions of knapsack problems are given. We present in section 3.4 a set of transformations between integer linear programs of the knapsack problems. In section 3.5, computational results are given to show the impact of these transformations. Finally, section 3.6 concludes this chapter.

3.2 Preliminaries

3.2.1 Integer linear program

Integer linear programming refers to mathematical programming with discrete variables and linearities in the objective function and constraints.

The general form of an integer linear program (ILP) is (see Figure 3.1)

**Figure 3.1:** A general form of an integer linear program

| maximize   | $c^T x$ |
| subject to | $Ax \leq b$ |
|           | $x \in X$ Integer |

where $c$ is a $n$-vector, $A$ is an $m \times n$ matrix and $b$ is an $m$-vector. $x$ is the decision variables required to be integer valued. $X$ is bounding-box-type restrictions on the variable. We refer to Chen et al. (2011) for more information about ILP fundamentals.
3.2.2 Knapsack problem constraints

The knapsack problem (KP) (Kellerer et al., 2004, Martello and Toth, 1990) is an integer linear program comprising binary variables, a single constraint with positive coefficients and binary restrictions on the variables.

The KP has been used to model various decision making processes and finds a variety of real world applications: resource allocation problems, cutting stock, capital budgeting, project selection and processor allocation in distributed computing systems. Industrial applications find the need for satisfying additional constraints such as multidimensional knapsack constraints, demand constraints and multiple choice constraints. These constraints can be defined as follows:

- **Knapsack constraint**: The knapsack constraint is to choose a subset of items set such that their overall weight does not exceed a knapsack capacity. In case when the knapsack has a set of dimensions, the constraint is called multidimensional knapsack constraint in which each dimension is called a knapsack constraint.

- **Demand constraint**: The demand constraint is to choose a subset of items set such that their overall weight must exceed a demand capacity. Like the above constraint, the demand constraint can be multidimensional.

- **Multiple choice constraint**: When items are distributed on a set of disjointed sets, the multiple choice constraint is to choose an item of each set.

These necessities (additional constraints) lead to many extensions and variants of knapsack problems such as the multidimensional knapsack problem (MKP) Freville (2004), the multiple demand knapsack problem (MDMKP) Cappanera and Trubian (2005), the multiple choice knapsack problem (MCKP) Pisinger (1995), the multidimensional multiple choice knapsack problem (MMKP) Moser et al. (1997a), Khan (1998) and multidimensional knapsack problems with generalized upper bound constraints (GUBMKP) Li (2005), Li and Curry (2005). In Table 3.1,
we characterize the problems mentioned above according to the type of constraints.

Table 3.1: Knapsack problems according to the type of constraints

<table>
<thead>
<tr>
<th>Problem</th>
<th>Knapsack</th>
<th>Multidimensional Knapsack</th>
<th>Multiple Demand</th>
<th>Multiple Choice</th>
</tr>
</thead>
<tbody>
<tr>
<td>KP</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MKP</td>
<td></td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>MDMKP</td>
<td></td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>MCKP</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MMKP</td>
<td></td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>GUBMKP</td>
<td></td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>MDMMKP</td>
<td></td>
<td></td>
<td></td>
<td>X</td>
</tr>
</tbody>
</table>

MDMMKP is an abbreviation of the multiple demand multidimensional multiple choice knapsack problem that will be defined in the next section and will considered as the generalized problem.

In order to give a more detailed presentation, let us define and denote the following terms (see Table 3.2).

Table 3.2: Notation of knapsack terms

- $N$: the number of items
- $p_j$: the profit of item $j$
- $w_j$: the weight of item $j$
- $c$: the capacity of a single knapsack
- $m$: the number of knapsack constraints
- $w_{jk}^i$: the weight of item $j$ in knapsack $k$
- $c_k^i$: the capacity of knapsack $k$
- $q$: the number of demand constraints
- $n$: the number of groups
- $G = \{G_1, \ldots, G_n\}$: the set of groups
- $|G_i|$: the number of items of group $G_i$
- $p_{ij}$: the profit of item $j$ of group $G_i$
- $w_{ij}^k$: the weight of item $j$ of group $G_i$ in knapsack $k$

3.2.3 Reduction, generalization and problems transformation

- Reduction and generalization
Given two related problems, such as A and B, we say that the problem A is a generalization of the problem B (denoted by B→ A) if and only if:

– every solution to problem A is also a solution to problem B; and
– there are solutions to problem B which are not solutions to problem A.

Note that the reduction is the symmetric relationship of the generalization. So, the problem B is a reduction of the problem A.

Besides it is obvious that the generalization and the reduction are a transitive relation. For example if B→ A and C→ B then C→ A.

• Problem transformation

We define a problem transformation as the operation that takes a problem and generates it into another problem in accordance with a set of rules without losing sight of specifications.

3.3 The knapsack problem family involving the notion of dimensions, demands and sets

3.3.1 The knapsack problem

The knapsack problem (KP) can be defined by a set of N items; each item j has a profit $p_j$ and a weight $w_j$. The problem is to choose a subset of the given items such that the corresponding total profit is maximized while the total weight satisfies the knapsack capacity $c$. It can be formulated as the following integer linear program (see Figure 3.2)

where Equation (1) provides the total profit of selecting items and Equation (2) ensures that the knapsack constraint is satisfied. The binary decision variables $x_j$ are used to indicate whether item j is included in the knapsack or not.

There follows (Figure 3.3) a small illustrative problem which will be used throughout the chapter. For this problem the number of items N is equal to 8.
3.3.2 The multidimensional knapsack problem

The multidimensional knapsack problem (MKP) Freville (2004) is considered as an extension of the classical knapsack problem in which knapsack has a set of dimensions. Each dimension is called a knapsack constraint.

The MKP can be defined by a set of $N$ items and a knapsack with $m$ dimensions. The knapsack has a limited capacity in each dimension $k$ denoted by $c^k$. Each item $j$ has a profit $p_j$ and a weight in each dimension, denoted by $w^k_j$. The goal is to select a subset of items with maximum total profit, see Equation (4). Chosen
items must, however, not exceed knapsack constraints, see Equation (5). The 0-1 decision variables \( x_j \) indicate which items are selected.

**Figure 3.4:** An integer linear program of the multidimensional knapsack problem

\[
\text{maximize } \sum_{j=1}^{N} p_j x_j \\
\text{subject to } \sum_{j=1}^{N} w_j^k x_j \leq c^k \quad (k = 1, \ldots, m) \\
x_j \in \{0, 1\} \quad (j = 1, \ldots, N)
\]  

Figure 3.5 represents an extended example of the knapsack problem to present the multidimensional knapsack problem where the number of dimensions \( m \) is equal to 2.

**Figure 3.5:** An instance example of the multidimensional knapsack problem

| Input: |  
|------|---|
| \( N \leftarrow 8 \)  \( m \leftarrow 2 \)  
| \( p_j \) | 10 20 30 40 50 60 70 80  
| \( w_j^k \) | 5 20 25 35 40 45 55 60 90 120 70 110 90 65 80 150 150 300  
| \( c^k \) |  
| Constraints: | \( 5x_1 + 20x_2 + 25x_3 + 35x_4 + 40x_5 + 45x_6 + 55x_7 + 60x_8 \leq 150 \)  
|  | \( 90x_1 + 120x_2 + 70x_3 + 110x_4 + 90x_5 + 65x_6 + 80x_7 + 150x_8 \leq 300 \)  
|  | \( x_j \in \{0, 1\} \quad (j = 1, \ldots, 8) \)  
| Objective: | maximize \( 10x_1 + 20x_2 + 30x_3 + 40x_4 + 50x_5 + 60x_6 + 70x_7 + 80x_8 \)  
| Optimal solution: | \( x_j \) | 0 0 0 0 0 0 1 1 | value 180 |
3.3.3 The multiple demand multidimensional knapsack problem

The multiple demand multidimensional knapsack problem (MDMKP) Cappanera and Trubian (2005) is considered as an extension of the multidimensional knapsack problem in which there are greater-than-or-equal-to inequalities called demand constraints, in addition to the standard less-than-or-equal-to inequalities. Formally the problem can be stated as integer linear program as shown in Figure 3.6.

Figure 3.6: An integer linear program of the multiple demand multidimensional knapsack problem

\[
\begin{align*}
\text{maximize} & \quad \sum_{j=1}^{N} p_j x_j \\
\text{subject to} & \quad \sum_{j=1}^{N} w_j^k x_j \leq c_k \quad (k = 1, \ldots, m) \\
& \quad \sum_{j=1}^{N} w_j^k x_j \geq c_k \quad (k = 1 + m, \ldots, (m + q)) \\
& \quad x_j \in \{0, 1\} \quad (j = 1, \ldots, N)
\end{align*}
\]  

Each of the \( m \) constraints of family Equation (8) represents a knapsack constraint, while each of the \( q \) constraints of family Equation (9) represents a demand constraint.

Figure 3.7 represents an example of the MDMKP which can be considered as an extension of the MKP example with two demand constraints \( q = 2 \).
Figure 3.7: An instance example of the multiple demand multidimensional knapsack problem

<table>
<thead>
<tr>
<th>Input:</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>$N \leftarrow 8$</td>
<td>$m \leftarrow 2$</td>
</tr>
<tr>
<td>$p_j$</td>
<td>10</td>
</tr>
<tr>
<td>$w^k_j$</td>
<td>5</td>
</tr>
<tr>
<td>90</td>
<td>120</td>
</tr>
<tr>
<td>5</td>
<td>20</td>
</tr>
<tr>
<td>90</td>
<td>60</td>
</tr>
<tr>
<td>$c^k$</td>
<td>150</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Constraints:</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>$5x_1 + 20x_2 + 25x_3 + 35x_4 + 40x_5 + 45x_6 + 55x_7 + 60x_8 \leq 150$</td>
<td></td>
</tr>
<tr>
<td>$90x_1 + 120x_2 + 70x_3 + 110x_4 + 90x_5 + 65x_6 + 80x_7 + 150x_8 \leq 300$</td>
<td></td>
</tr>
<tr>
<td>$5x_1 + 20x_2 + 100x_3 + 35x_4 + 60x_5 + 45x_6 + 50x_7 + 60x_8 \geq 80$</td>
<td></td>
</tr>
<tr>
<td>$90x_1 + 60x_2 + 70x_3 + 110x_4 + 90x_5 + 45x_6 + 20x_7 + 10x_8 \geq 200$</td>
<td></td>
</tr>
<tr>
<td>$x_j \in {0, 1}$ ($j = 1, \ldots, 8$)</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Objective:</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>maximize $10x_1 + 20x_2 + 30x_3 + 40x_4 + 50x_5 + 60x_6 + 70x_7 + 80x_8$</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Optimal solution:</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>$x_j$</td>
<td>0</td>
</tr>
</tbody>
</table>

3.3.4 The multiple choice knapsack problem

The multiple choice knapsack problem (MCKP) Pisinger (1995) is considered as an extension of the classical knapsack problem in which items are distributed on $n$ disjointed groups $G = (G_1 \cup G_2 \cup \ldots \cup G_n)$ (see Equation 11).

$$\forall (p, q), p \neq q, p \leq n, q \leq n, \ G_p \cap G_q = \emptyset \text{ and } \bigcup_{i=1}^{n} G_i = G \quad (11)$$

The MCKP consists in selecting one and only one item of each group without violating the knapsack capacity $c$ in order to maximize the total profit of the selected items. The MCKP can be modeled as an integer linear program as shown in Figure 3.8.
Chapter 2. The multiple demand multidimensional multiple choice KP

Figure 3.8: An integer linear program of the multiple choice knapsack problem

\[
\begin{align*}
\text{maximize} & \quad \sum_{i=1}^{n} \sum_{j=1}^{|G_i|} p_{ij} x_{ij} \\
\text{subject to} & \quad \sum_{i=1}^{n} \sum_{j=1}^{|G_i|} w_{ij} x_{ij} \leq c \\
& \quad \sum_{j=1}^{|G_i|} x_{ij} = 1 \quad (i = 1, \ldots, n) \\
& \quad x_{ij} \in \{0, 1\} \quad (i = 1, \ldots, n, (j = 1, \ldots, |G_i|) \\
\end{align*}
\] (12)

(13)

(14)

(15)

The variable \(x_{ij}\) is equal to 1 when the item \(j\) of the group \(G_i\) is selected, 0 otherwise. The objective function Equation (12) represents the total profit to be maximized. The knapsack constraint is presented in Equation (13) and the \(n\) multiple choice constraints are presented in Equation (14). To avoid unsolvable situations, we assume that the sum of the minimum weight of items in each group is smaller than the knapsack capacity \(c\).

Figure 3.9 represents an example of MCKP where items are subdivided into \(n = 3\) groups, the cardinality of \(G_1 = 3\), \(G_2 = 2\) and \(G_3 = 3\).

3.3.5 The multidimensional multiple choice knapsack problem

The multidimensional multiple choice knapsack problem (MMKP) Moser et al. (1997a), Khan (1998) is a particular variant of the knapsack problem. It can be viewed as a combination of aspects between the multidimensional knapsack problem (MKP) and the multiple choice knapsack problem (MCKP). The MMKP is an extension of the MCKP in which one item is selected from each group. However, in the MMKP, the knapsack is multidimensional, i.e., the knapsack consists of multiple resource constraints simultaneously satisfied. The MMKP problem can be stated as an integer linear program as shown in Figure 3.10.

Equation (16) provides the profit of selecting items, a value to be maximized. Equation (17) ensures the resource capacity of knapsack \(k\) is not exceeded while
Figure 3.9: An instance example of the multiple choice knapsack problem

**Input:**

\[ N \leftarrow 8 \quad n \leftarrow 3 \]

\[ p_{ij} \quad G_1: 10 \quad 20 \quad 30 \quad G_2: 40 \quad 50 \quad 60 \quad 70 \quad 80 \quad G_3 \]

\[ w_{ij} \quad G_1: 5 \quad 20 \quad 25 \quad G_2: 35 \quad 40 \quad 45 \quad 55 \quad 60 \quad G_3 \]

\[ c \quad 130 \]

**Constraints:**

\[ 5x_{11} + 20x_{12} + 25x_{13} + 35x_{21} + 40x_{22} + 45x_{31} + 55x_{32} + 60x_{33} \leq 130 \]

\[ x_{11} + x_{12} + x_{13} = 1 \]

\[ x_{21} + x_{22} = 1 \]

\[ x_{31} + x_{32} + x_{33} = 1 \]

\[ x_{ij} \in \{0, 1\} \quad (i = 1, 2, 3) \quad (j = 1, \ldots, |G_i|) \]

**Objective:**

\[ \text{maximize} \quad 10x_{11} + 20x_{12} + 30x_{13} + 40x_{21} + 50x_{22} + 60x_{31} + 70x_{32} + 80x_{33} \]

**Optimal solution:**

\[ 0 \quad 1 \quad 0 \quad G_1 \quad 0 \quad 1 \quad G_2 \quad 0 \quad 0 \quad 1 \quad G_3 \quad \text{value} \quad 150 \]

Figure 3.10: An integer linear program of the multidimensional multiple choice knapsack problem

\[
\begin{align*}
\text{maximize} & \quad \sum_{i=1}^{n} \sum_{j=1}^{|G_i|} p_{ij}x_{ij} \\
\text{subject to} & \quad \sum_{i=1}^{n} \sum_{j=1}^{|G_i|} w_{ij}^{k}x_{ij} \leq c^{k} \quad (k = 1, \ldots, m) \\
& \quad \sum_{j=1}^{|G_i|} x_{ij} = 1 \quad (i = 1, \ldots, n) \\
& \quad x_{ij} \in \{0, 1\} \quad (i = 1, \ldots, n), (j = 1, \ldots, |G_i|)
\end{align*}
\]

Equation (18) ensures selecting a single item from each of the \( G_i \) groups. Equation (19) is the binary selection requirement on decision variable \( x_{ij} \) such that \( x_{ij} \) is equal to 1 if the item \( j \) of the group \( G_i \) is selected, 0 otherwise.
Chapter 2. The multiple demand multidimensional multiple choice KP

Figure 3.11 represents an extended example of the multiple choice knapsack problem to present the multidimensional multiple choice knapsack problem where the number of dimensions \( m \) is equal to 2.

**Figure 3.11:** An instance example of the multidimensional multiple choice knapsack problem

| Input: |  
|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|
| \( N \) ← 8 \( n \) ← 3 \( m \) ← 2 | → **Profits** → | \( p_{ij} \) | \( G_1 \) | \( G_2 \) | \( G_3 \) | \( G_1 \) | \( G_2 \) | \( G_3 \) |
| | | 10 | 20 | 30 | 40 | 50 | 60 | 70 | 80 |
| Constraints: | → **Weights** → | \( w_{ij} \) | \( G_1 \) | \( G_2 \) | \( G_3 \) | \( G_1 \) | \( G_2 \) | \( G_3 \) |
| | | 5 | 10 | 12 | 20 | 25 | 35 | 40 | 45 | 55 | 60 |
| Objective: | → **Capacities** → | \( c_k \) | 130 | 230 |
| | | \( x_{ij} \) ∈ \{0, 1\} (\( i = 1, 2, 3 \) \( j = 1, \ldots, |G_i| \)) | \( x_{11} + x_{12} + x_{13} = 1 \) | \( x_{21} + x_{22} = 1 \) | \( x_{31} + x_{32} + x_{33} = 1 \) | maximize | \( 10x_{11} + 20x_{12} + 30x_{13} + 40x_{21} + 50x_{22} + 60x_{31} + 70x_{32} + 80x_{33} \) | \( 0 \ 0 \ 1 \) | \( 0 \ 1 \) | \( 1 \ 0 \ 0 \) | value | 140 |

### 3.3.6 The multidimensional knapsack problems with generalized upper bound constraints

The multidimensional knapsack problem with generalized upper bound constraints (GUBMKP) Li (2005), Li and Curry (2005) is defined as a multidimensional knapsack problem (MKP) with mutually exclusive generalized upper-bound(GUB) constraints where all GUBs are fixed at 1. It can be viewed as a reduction of the
MMKP, in which it is required that at most one item per group can be chosen. The GUBMKP problem is formulated as an integer linear program as shown in Figure 3.12.

**Figure 3.12:** An integer linear program of the multidimensional knapsack problems with generalized upper bound constraints

\[
\begin{align*}
\text{maximize} & \quad \sum_{i=1}^{n} \sum_{j=1}^{|G_i|} p_{ij} x_{ij} \\
\text{subject to} & \quad \sum_{i=1}^{n} \sum_{j=1}^{|G_i|} w_{ij} x_{ij} \leq c_k \quad (k = 1, \ldots, m) \quad (21) \\
& \quad \sum_{j=1}^{|G_i|} x_{ij} \leq 1 \quad (i = 1, \ldots, n) \quad (22) \\
& \quad x_{ij} \in \{0, 1\} \quad (i = 1, \ldots, n), (j = 1, \ldots, |G_i|) \quad (23)
\end{align*}
\]

Equation (20) represents the total profit to be maximized. Equation (21) ensures the knapsack capacities are not exceeded while Equation (22) ensures selecting at most one item from each of the \(n\) disjoint groups. Equation (23) is the binary selection requirement on decision variable \(x_{ij}\) such that \(x_{ij}\) is equal to 1 when the item \(j\) of the group \(G_i\) is selected, 0 otherwise.

Since the structure and the specification of the GUBMKP is similar to the MMKP, we use the same input of the latter to present an example of the GUBMKP (see Figure 3.13)

### 3.3.7 The multiple demand multidimensional multiple choice knapsack problem

We define the multiple demand multidimensional multiple choice knapsack problem (MDMMKP) as a combination of aspects of the multidimensional knapsack constraint, multiple demand constraint and multiple choice constraint. It is considered as an extension of the multidimensional multiple choice knapsack problem (MMKP) in which there are greater-than-or-equal-to inequalities, in addition to the standard less-than-or-equal-to inequalities. The integer linear program of the MDMMKP can be stated in Figure 3.14.
Figure 3.13: An instance example of the multidimensional knapsack problems with generalized upper bound constraints

**Constraints:**

\[
\begin{align*}
5x_{11} + 20x_{12} + 25x_{13} + 35x_{21} + 40x_{22} + 45x_{31} + 55x_{32} + 60x_{33} & \leq 130 \\
90x_{11} + 120x_{12} + 70x_{13} + 110x_{21} + 90x_{22} + 65x_{31} + 80x_{32} + 150x_{33} & \leq 230 \\
x_{11} + x_{12} + x_{13} & \leq 1 \\
x_{21} + x_{22} & \leq 1 \\
x_{31} + x_{32} + x_{33} & \leq 1 \\
x_{ij} & \in \{0, 1\} \quad (i = 1, 2, 3) \quad (j = 1, \ldots, |G_i|)
\end{align*}
\]

**Objective:**

\[
\text{maximize} \quad 10x_{11} + 20x_{12} + 30x_{13} + 40x_{21} + 50x_{22} + 60x_{31} + 70x_{32} + 80x_{33}
\]

**Optimal solution:**

\[
\begin{array}{ccc}
0 & 0 & 1 \\
0 & 1 & 0 \\
1 & 0 & 0
\end{array} \quad \text{value} \quad 140
\]

Figure 3.14: An integer linear program of the multidimensional multiple demand multiple choice knapsack problem

\[
\begin{align*}
\text{maximize} & \quad \sum_{i=1}^{n} \sum_{j=1}^{|G_i|} p_{ij} x_{ij} \\
\text{subject to} & \quad \sum_{i=1}^{n} \sum_{j=1}^{|G_i|} w_{ij} x_{ij} \leq c^k \quad (k = 1, \ldots, m) \tag{25} \\
& \quad \sum_{i=1}^{n} \sum_{j=1}^{|G_i|} w_{ij} x_{ij} \geq c^k \quad (k = 1 + m, \ldots, (m + q)) \tag{26} \\
& \quad \sum_{j=1}^{|G_i|} x_{ij} = 1 \quad (i = 1, \ldots, n) \tag{27} \\
x_{ij} & \in \{0, 1\} \quad (i = 1, \ldots, n), (j = 1, \ldots, |G_i|) \tag{28}
\end{align*}
\]

The variable \(x_{ij}\) is equal to 1 when the item \(j\) of the group \(G_i\) is selected, 0 otherwise. The objective function equation (24) represents the total profit to be maximized. The knapsack constraints are presented in equation (25) and the demand constraints are presented in equation (26). Equation (27) represents the \(n\) multiple choice constraints.

There follows an illustrative example that represents an extension of the MMKP example with \(q = 2\) demand constraints.
Input:

\[ N \leftarrow 8 \quad n \leftarrow 3 \quad m \leftarrow 2 \]

\[ \rightarrow \text{Profits} \rightarrow \]

\[ \begin{array}{c|c|c}
   & G_1 & G_2 \\
   p_{ij} & 10 & 30 \\
   & 40 & 50 \\
   & 60 & 70 \\
   & 80 &
\end{array} \]

\[ \rightarrow \text{Weights} \rightarrow \]

\[ \begin{array}{c|c|c|c|c|c|c|c|c}
   & G_1 & G_2 & G_3 \\
   w_{ij} & 5 & 20 & 25 & 35 & 40 & 45 & 55 & 60 \\
   & 90 & 120 & 70 & 110 & 90 & 65 & 80 & 150 \\
   & 5 & 20 & 100 & 35 & 60 & 45 & 50 & 60 \\
   & 90 & 60 & 70 & 110 & 90 \\
   & 45 & 20 & 10 &
\end{array} \]

\[ \rightarrow \text{Capacities} \rightarrow \]

\[ c^k \]

Constraints:

\[ 5x_{11} + 20x_{12} + 25x_{13} + 35x_{21} + 40x_{22} + 45x_{31} + 55x_{32} + 60x_{33} \leq 130 \]
\[ 90x_{11} + 120x_{12} + 70x_{13} + 110x_{21} + 90x_{22} + 65x_{31} + 80x_{32} + 150x_{33} \leq 230 \]
\[ 5x_{11} + 20x_{12} + 25x_{13} + 35x_{21} + 40x_{22} + 45x_{31} + 55x_{32} + 60x_{33} \geq 80 \]
\[ 90x_{11} + 60x_{12} + 70x_{13} + 110x_{21} + 90x_{22} + 45x_{31} + 20x_{32} + 10x_{33} \geq 200 \]
\[ x_{11} + x_{12} + x_{13} = 1 \]
\[ x_{21} + x_{22} = 1 \]
\[ x_{31} + x_{32} + x_{33} = 1 \]
\[ x_{ij} \in \{0, 1\} \quad (i = 1, 2, 3) \quad (j = 1, \ldots, |G_i|) \]

Objective:

\[ \text{maximize} \quad 10x_{11} + 20x_{12} + 30x_{13} + 40x_{21} + 50x_{22} + 60x_{31} + 70x_{32} + 80x_{33} \]

Optimal solution:

\[ \begin{array}{c|c|c|c}
   & G_1 & G_2 & G_3 \\
   0 & 0 & 1 &
\end{array} \quad \begin{array}{c|c|c|c}
   & G_1 & G_2 &
\end{array} \quad \begin{array}{c|c|c|c}
   & G_1 & G_2 &
\end{array} \quad \text{value} \quad 140 \]

3.3.8 Relation schema between problems

Figure 3.16 shows a set of possible generalizations between the problems mentioned above where each arrow represents a generalization between a target problem and a destination problem. In fact:

- The problems in which knapsack constraint is multidimensional are a generalization of problems with one dimension. Among these problems the MKP
Chapter 2. The multiple demand multidimensional multiple choice KP

and the MMKP that are considered as a generalization of KP and MCKP respectively.

- The problems with demand constraints are a generalization of the problems without demand constraints. It is apparent that the MDMKP and the MDMMKP are considered as a generalization of the MKP and MMKP respectively.

- The problems that have a notion of groups such as MCKP, MMKP and MDMMKP are a generalization of problems without groups such as KP, MKP and MDMKP respectively.

- The GUBMKP is a generalization of MKP and a reduction of MMKP.

Figure 3.16: Relation between knapsack problems

According to the transitivity characteristics of the generalization between problems we consider the MDMMKP as the most generalized problem.

3.4 Transformations between Integer Linear Programs

In this section, we present a set of transformations between the different integer linear programs of the knapsack problems mentioned above. These transformations are summarized in Figure 3.17. Each arrow indicates that the transformation
between the linked problems is proved.

### 3.4.1 Transformation of the GUBMKP into the MMKP

The GUBMKP can be easily transformed into the MMKP. The formulation of the MMKP can be built by substituting the inequality (≤) by the strict equality (=) for equation (22) in the GUBMKP formulation. Variables in each of the multiple choice constraints sum to 1 exactly. This modification is obtained by adding a dummy item into each group (GUB constraint) in which its consumption and its profit are zero. Indeed, selecting any item in the original GUBMKP formulation is similar to selecting the dummy item in the generated MMKP formulation. Therefore, the GUBMKP formulation mentioned above can be transformed into the MMKP formulation as shown in Figure 3.18.

**Figure 3.18:** An integer linear program of GUBMKP based on the MMKP formulation

\[
\text{maximize} \quad \sum_{i=1}^{n} \sum_{j=1}^{|G_i|+1} p_{ij} x_{ij} 
\]

subject to

\[
\sum_{i=1}^{n} \sum_{j=1}^{|G_i|+1} w_{ij} x_{ij} \leq c_k \quad (k = 1, \ldots, m) \quad (30)
\]

\[
\sum_{j=1}^{|G_i|+1} x_{ij} = 1 \quad (i = 1, \ldots, n) \quad (31)
\]

\[
x_{ij} \in \{0, 1\} \quad (i = 1, \ldots, n), (j = 1, \ldots, |G_i| + 1) \quad (32)
\]
where \( p_{i(|G_i|+1)} = 0 \) and \( w_{i(|G_i|+1)} = 0 \) (\( i = 1, \ldots, n \)).

The instance example of the GUBMKP (Figure 3.13) is moved to an MMKP instance (see Figure 3.19)

**Figure 3.19:** An instance example of the GUBMKP based on the MMKP formulation

<table>
<thead>
<tr>
<th>Input:</th>
<th>N ← 11  n ← 3  m ← 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>( N )</td>
<td>( n )</td>
</tr>
<tr>
<td>→ Profits →</td>
<td>→ Weights →</td>
</tr>
<tr>
<td>( p_{ij} )</td>
<td>( w_{ij} )</td>
</tr>
<tr>
<td>( G_1 )</td>
<td>( G_2 )</td>
</tr>
<tr>
<td>10 20 30 0</td>
<td>40 50 0</td>
</tr>
<tr>
<td>( G_1 )</td>
<td>( G_2 )</td>
</tr>
<tr>
<td>( x_{ij} \in {0, 1} ) (( i = 1, 2, 3 )) (( j = 1, \ldots,</td>
<td>G_i</td>
</tr>
<tr>
<td>Constraints:</td>
<td>Constraints:</td>
</tr>
<tr>
<td>( 5x_{11} + 20x_{12} + 25x_{13} + 35x_{21} + 40x_{22} + 45x_{31} + 55x_{32} + 60x_{33} \leq 130 )</td>
<td>( 90x_{11} + 120x_{12} + 70x_{13} + 110x_{21} + 90x_{22} + 65x_{31} + 80x_{32} + 150x_{33} \leq 230 )</td>
</tr>
<tr>
<td>( x_{21} + x_{22} + x_{23} = 1 )</td>
<td>( x_{31} + x_{32} + x_{33} + x_{34} = 1 )</td>
</tr>
<tr>
<td>Objective:</td>
<td>Objective:</td>
</tr>
<tr>
<td>maximize ( 10x_{11} + 20x_{12} + 30x_{13} + 40x_{21} + 50x_{22} + 60x_{31} + 70x_{32} + 80x_{33} )</td>
<td></td>
</tr>
<tr>
<td>Optimal solution:</td>
<td>Optimal solution:</td>
</tr>
<tr>
<td>( G_1 )</td>
<td>( G_2 )</td>
</tr>
<tr>
<td>0 0 1 0</td>
<td>0 1 0</td>
</tr>
<tr>
<td>value</td>
<td>140</td>
</tr>
</tbody>
</table>

### 3.4.2 Transformation of the MKP into the MMKP

To explain this part, let us take the ILP of the MKP mentioned above. We can transform a MKP instance into a MMKP instance by creating a set of \( n \) groups, each one contains two items. The first item is an item of the MKP and the second item presents a dummy item whose weight and profit are zero. Consequently, each
Chapter 2. *The multiple demand multidimensional multiple choice KP*

group $G_j$ ($j = 1, \ldots, n$) contains two items where:

$$p_{j1} = p_j, \quad p_{j2} = 0, \quad w_{j1}^k = w_j^k, \quad w_{j2}^k = 0 \quad (j = 1, \ldots, n) \quad (k = 1, \ldots, m)$$

For example, in case when the item is not selected in the original formulation of MKP, it is similar to select the dummy item belonging to the same set in the MMKP formulation.

The integer linear program of the MKP can be transformed into an integer linear program of the MMKP as shown in Figure 3.20.

**Figure 3.20:** An integer linear program of MKP based on the MMKP formulation

$$\text{maximize} \quad \sum_{i=1}^{N} \sum_{j=1}^{2} p_{ij} x_{ij} \quad (33)$$

subject to

$$\sum_{i=1}^{N} \sum_{j=1}^{2} w_{ij}^k x_{ij} \leq c_k \quad ((k = 1, \ldots, m) \quad (34))$$

$$\sum_{j=1}^{2} x_{ij} = 1 \quad (i = 1, \ldots, N) \quad (35)$$

$$x_{ij} \in \{0, 1\} \quad (i = 1, \ldots, N), (j = 1, 2) \quad (36)$$

The instance example of the MKP (Figure 3.5) is moved to an MMKP instance (see Figure 3.21)

Note that we will use the same principle to move the MDMKP into the MDMMKP and the KP into the MCKP.

### 3.4.3 Transformation of the GUBMKP into the MKP

In this part, we are going to transform the GUBMKP into the MKP. Let us define the following terms $N = \sum_{i=1}^{n} |G_i|$, $N_h = \sum_{i=1}^{h} |G_i|$, $\forall h \in (1, \ldots, n)$ and $N_0 = 0$, and rename the following terms $x_{ij} = y_l$ and $w_{ij}^k = w_l^k$ where $l = N_{i-1} + j$.

The idea in this part is to view each GUB constraint as a knapsack constraint. So we add to the classical $m$ knapsack constraints a new $n$ constraints by substituting
Figure 3.21: An instance example of the MKP based on the MMKP formulation

Input:
\( \begin{align*} N & \leftarrow 16 \quad m \leftarrow 2 \\
p_j & \begin{pmatrix} 10 & 20 & 30 & 40 & 50 & 60 & 70 & 80 \\
g_1 & g_2 & g_3 & g_4 & g_5 & g_6 & g_7 & g_8 \end{pmatrix} \\
\rightarrow Profits \rightarrow \\
w_k^j & \begin{pmatrix} 5 & 0 & 20 & 0 & 25 & 0 & 35 & 0 & 40 & 0 & 45 & 0 & 55 & 0 & 60 & 0 \\
g_1 & g_2 & g_3 & g_4 & g_5 & g_6 & g_7 & g_8 \end{pmatrix} \\
\rightarrow Weights \rightarrow \\
c^k & \\
& \begin{pmatrix} 150 & 300 \end{pmatrix} \\
\rightarrow Capacity \rightarrow \\
\end{align*} \)

Constraints:
\( \\
5x_{11} + 20x_{21} + 25x_{31} + 35x_{41} + 40x_{51} + 45x_{61} + 55x_{71} + 60x_{81} \leq 150 \\
90x_{11} + 120x_{21} + 70x_{31} + 110x_{41} + 90x_{51} + 65x_{61} + 80x_{71} + 150x_{81} \leq 300 \\
x_{11} + x_{12} = 1 \\
x_{21} + x_{22} = 1 \\
x_{31} + x_{32} = 1 \\
x_{41} + x_{42} = 1 \\
x_{51} + x_{52} = 1 \\
x_{61} + x_{62} = 1 \\
x_{71} + x_{72} = 1 \\
x_{81} + x_{82} = 1 \\
x_{ij} \in \{0, 1\} \quad (i = 1, \ldots, 8) \quad (j = \{1, 2\}) \\
\end{align*} \)

Objective:
\( max \ 10x_{11} + 20x_{21} + 30x_{31} + 40x_{41} + 50x_{51} + 60x_{61} + 70x_{71} + 80x_{81} \)

Optimal solution:
\( \\
\begin{pmatrix} 0 & 1 \\
g_1 & g_2 \end{pmatrix} \quad \begin{pmatrix} 1 & 0 \\
g_3 & g_4 \end{pmatrix} \quad \begin{pmatrix} 0 & 1 \\
g_5 & g_6 \end{pmatrix} \quad \begin{pmatrix} 0 & 1 \\
g_7 & g_8 \end{pmatrix} \quad value \quad 180 \\
\end{pmatrix} \)

the \( n \) GUB constraints to \( n \) equivalent knapsack constraints. In fact the ILP of the GUBMKP can be reformulated to an ILP of the MKP in this manner:

The \( n \) GUBMKP constraints \( \sum_{j=1}^{[G_i]} x_{ij} \leq 1, \ \forall i \in (1, \ldots, n) \) are transformed into \( \sum_{i=1}^{N} w_i^k y_i \leq c^k, \ \forall k \in (m + 1, \ldots, m + n) \) where \( c^k = 1 \ \forall k \in (m + 1, \ldots, m + n) \) and for each \( k + m \) knapsack constraint \( \forall k \in (1, \ldots, n) \)
Chapter 2. The multiple demand multidimensional multiple choice KP

Figure 3.22: An integer linear program of MKP based on the MMKP formulation

\[
\begin{align*}
\text{maximize} & \quad \sum_{l=1}^{N} p_l y_l \\
\text{subject to} & \quad \sum_{l=1}^{N} w^k_l y_l \leq c^k \quad (k = 1, \ldots, m+n) \\
& \quad y_l \in \{0, 1\} \quad (l = 1, \ldots, N)
\end{align*}
\]

The instance example of the GUBMKP (Figure 3.13) is moved to an MKP instance (see Figure 3.23)

3.4.4 Transformation of the MMKP into the MDMKP

The \(n\) multiple choice constraints of the MMKP can be subdivided into two kinds of constraints \(n\) less-than-or-equal-to and \(n\) greater-than-or-equal-to inequalities. Indeed, the MMKP is modeled as shown in Figure 3.24.

And using the same principle to transform the GUBMKP into the MKP mentioned above, we can view the MMKP program as an MDMKP program as shown in Figure 3.25.

where \(c^k = 1 \forall k \in (m+1, \ldots, m+n, m+n+1, \ldots, m+2n)\).

The instance example of the MMKP (Figure 3.11) is moved to an MDMKP instance (see Figure 3.26)

3.4.5 Transformation of the MCKP into the GUBMKP

The MCKP can be transformed into an equivalent GUBMKP. The transformation is done by removing one of the items from each group. We denote this item \(j^*\)
Figure 3.23: An instance example of the GUBMKP based on the MKP formulation

<table>
<thead>
<tr>
<th>Input:</th>
</tr>
</thead>
<tbody>
<tr>
<td>$N \leftarrow 8$</td>
</tr>
<tr>
<td>$m \leftarrow 5$</td>
</tr>
<tr>
<td>$p_j$</td>
</tr>
<tr>
<td>$w_j^k$</td>
</tr>
<tr>
<td>90</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>0</td>
</tr>
<tr>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Constraints:</th>
</tr>
</thead>
<tbody>
<tr>
<td>$5x_1 + 20x_2 + 25x_3 + 35x_4 + 40x_5 + 45x_6 + 55x_7 + 60x_8 \leq 150$</td>
</tr>
<tr>
<td>$90x_1 + 120x_2 + 70x_3 + 110x_4 + 90x_5 + 65x_6 + 80x_7 + 150x_8 \leq 300$</td>
</tr>
<tr>
<td>$x_1 + x_2 + x_3 \leq 1$</td>
</tr>
<tr>
<td>$x_4 + x_5 \leq 1$</td>
</tr>
<tr>
<td>$x_6 + x_7 + x_8 \leq 1$</td>
</tr>
<tr>
<td>$x_j \in {0, 1}$ (for $j = 1, \ldots, 8$)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Objective:</th>
</tr>
</thead>
<tbody>
<tr>
<td>maximize $10x_1 + 20x_2 + 30x_3 + 40x_4 + 50x_5 + 60x_6 + 70x_7 + 80x_8$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Optimal solution:</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x_j$</td>
</tr>
</tbody>
</table>

where $x_{ij^*}$ is the variable with $w_{ij^*} = \min\{w_{ij} \forall j \in (1, \ldots, |G_i|)\}$, $\forall i \in (1, \ldots, n)$. For purposes of explanation, we modify the index of items in order that the index $j^*$ of each group $G_i$ is equal to $|G_i| \forall i \in (1, \ldots, n)$. Then the MCKP can be modeled as shown in Figure 3.27.

where $\bar{v} = \sum_{i=1}^{n} p_{ij^*}$. This value presents a lower bound of the MCKP, $c' = c - \sum_{i=1}^{n} w_{ij^*}$, $p'_{ij} = p_{ij} - p_{ij^*}$ and $w'_{ij} = w_{ij} - w_{ij^*}$.

With this kind of modeling, we are sure that at most one item is selected in each group. This item represents the item which has the smallest weight for each group denoted by $j^*$. So if any item of group $G_i$ is selected on the GUBMKP formulation, we are sure that the item $j^*$ is selected.
Figure 3.24: A second integer linear program of MMKP

\[
\text{maximize} \quad \sum_{i=1}^{n} \sum_{j=1}^{|G_i|} p_{ij} x_{ij} \quad (40)
\]

subject to

\[
\sum_{i=1}^{n} \sum_{j=1}^{|G_i|} w_{ij} x_{ij} \leq c_k \quad (k = 1, \ldots, m) \quad (41)
\]
\[
\sum_{j=1}^{|G_i|} x_{ij} \leq 1 \quad (i = 1, \ldots, n) \quad (42)
\]
\[
\sum_{j=1}^{|G_i|} x_{ij} \geq 1 \quad (i = 1, \ldots, n) \quad (43)
\]
\[
x_{ij} \in \{0, 1\} \quad (i = 1, \ldots, n), (j = 1, \ldots, |G_i|) \quad (44)
\]

Figure 3.25: An integer linear program of MMKP based on the MDMKP formulation

\[
\text{maximize} \quad \sum_{l=1}^{N} p_l y_l \quad (45)
\]

subject to

\[
\sum_{l=1}^{N} w_{lk} y_l \leq c_k \quad (k = 1, \ldots, m + n) \quad (46)
\]
\[
\sum_{l=1}^{N} w_{lk} y_l \geq c_k, \quad (k = m + n + 1, \ldots, m + 2n) \quad (47)
\]
\[
y_l \in \{0, 1\} \quad (l = 1, \ldots, N) \quad (48)
\]

The instance example of the MCKP (Figure 3.9) is moved to a GUBMKP instance (see Figure 3.28)

### 3.4.6 Algorithms of MDMMKP are able to solve the other problems

Given that any algorithm can solve the MDMMKP, based on the transitivity characteristics of the generalization and the set of transformations, this algorithm is able to solve all the other problems. In fact, we can easily transform their instances into the MDMMKP ones which can be solved directly by the mentioned algorithm.
3.5 Experimental results

This section is aimed at:

- validating experimentally the transformations between problems.
Chapter 2. The multiple demand multidimensional multiple choice KP

Figure 3.27: An integer linear program of MCKP based on the GUBMKP formulation

\[ \text{maximize } \bar{v} + \sum_{i=1}^{n} \sum_{j=1}^{|G_i|-1} p'_{ij} x_{ij} \]

subject to

\[ \sum_{i=1}^{n} \sum_{j=1}^{|G_i|-1} w'_{ij} x_{ij} \leq c' \] (50)
\[ \sum_{j=1}^{|G_i|-1} x_{ij} \leq 1 \quad (i = 1, \ldots, n) \] (51)
\[ x_{ij} \in \{0, 1\} \quad (i = 1, \ldots, n) \quad (j = 1, \ldots, |G_i| - 1) \] (52)

- answering the following questions: Are the generated problems using the transformations able to produce a reasonable CPU time value compared with the original formulations?

Note that the computational platform used to solve the test problems is consisted as the CPLEX Solver version 9.0 on a Windows XP with 2.50 GHz and 2 GB of shared memory. We modeled the MKP, MDKP, MMKP, GUBMKP and MD-MMKP into CPLEX.

3.5.1 Instances details

To test the set of transformations mentioned in the last section, we use a set of benchmarks available in OR-LIBRARY Beasley (1990) maintained by Beasley. The proposed transformation of the MKP into the MMKP is tested on sets of MKP instances. Indeed, these instances are available at OR-LIBRARY and results have been published by Chu and Beasley (1998). The problem instances that we considered are summarized in Table 3.3, where each instance set mknapcb for \( i = (1, \ldots, 6) \) contains 30 instances. The headers Set, \( N \) and \( m_1 \) indicate the name of the set, the number of items in each instance and the number of knapsack constraints respectively.

For the MDMKP, we use the benchmarks proposed by Cappanera and Trubian (2005). These instances are generated by properly modifying the MKP instances
Figure 3.28: An instance example of the MCKP based on the GUBMKP formulation

<table>
<thead>
<tr>
<th>Input:</th>
</tr>
</thead>
<tbody>
<tr>
<td>$N \leftarrow 8$  $n \leftarrow 3$</td>
</tr>
<tr>
<td>$\quad \rightarrow \text{Profits} \rightarrow$</td>
</tr>
<tr>
<td>$p_{ij}$</td>
</tr>
<tr>
<td>$\quad \rightarrow \text{Weights} \rightarrow$</td>
</tr>
<tr>
<td>$w_{ij}$</td>
</tr>
<tr>
<td>$\rightarrow \text{Capacities} \rightarrow$</td>
</tr>
<tr>
<td>$c$</td>
</tr>
</tbody>
</table>

Constraints:

$15x_{11} + 20x_{12} + 5x_{21} + 10x_{31} + 15x_{32} \leq 45$

$x_{11} + x_{12} \leq 1$

$x_{21} \leq 1$

$x_{31} + x_{32} \leq 1$

$x_{ij} \in \{0, 1\} \quad (i = 1, 2, 3) \quad (j = 1, \ldots, |G_i|)$

Objective:

$maximize$  $110 = (10 + 40 + 60) + 10x_{11} + 20x_{12} + 10x_{21} + 10x_{31} + 20x_{32}$

Optimal solution:

$\quad \quad \quad \frac{1}{G_1} \quad \frac{0}{G_2} \quad \frac{1}{G_3}$ \quad \text{value} \quad 150$

Table 3.3: Test MKP instances details

<table>
<thead>
<tr>
<th>Set</th>
<th>$N$</th>
<th>$m_1$</th>
</tr>
</thead>
<tbody>
<tr>
<td>mknapcb1</td>
<td>100</td>
<td>5</td>
</tr>
<tr>
<td>mknapcb2</td>
<td>250</td>
<td>5</td>
</tr>
<tr>
<td>mknapcb3</td>
<td>500</td>
<td>5</td>
</tr>
<tr>
<td>mknapcb4</td>
<td>100</td>
<td>10</td>
</tr>
<tr>
<td>mknapcb5</td>
<td>250</td>
<td>10</td>
</tr>
<tr>
<td>mknapcb6</td>
<td>500</td>
<td>10</td>
</tr>
</tbody>
</table>

solved in Chu and Beasley (1998). Given an MKP instance with $m_1$ knapsack constraints, 6 MDMKP instances are generated, one for each combination of profits type (either positive or mixed) and number of constraints ($m_2 = 1$, $m_2 = m_1/2$ and $m_2 = m$ respectively).

We test the first six instances sets where each set $mdmkp_{ct}$ for $i = (1, \ldots, 6)$
contains 90 instances. The MDKP instances are reported in Table 3.4 where the headers Set, \( N \), \( m_1 \) and \( m_2 \) indicates the name of the set, the number of items in each set, the number of knapsack constraints, and the number of demand constraints respectively.

\textbf{Table 3.4: Test MDMKP instances details}

<table>
<thead>
<tr>
<th>Set</th>
<th>( N )</th>
<th>( m_1 )</th>
<th>( m_2 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>mdkp_ct1</td>
<td>100</td>
<td>5</td>
<td>1 2 5</td>
</tr>
<tr>
<td>mdkp_ct2</td>
<td>250</td>
<td>5</td>
<td>1 2 5</td>
</tr>
<tr>
<td>mdkp_ct3</td>
<td>500</td>
<td>5</td>
<td>1 2 10</td>
</tr>
<tr>
<td>mdkp_ct4</td>
<td>100</td>
<td>10</td>
<td>1 5 10</td>
</tr>
<tr>
<td>mdkp_ct5</td>
<td>250</td>
<td>10</td>
<td>1 5 10</td>
</tr>
<tr>
<td>mdkp_ct6</td>
<td>500</td>
<td>10</td>
<td>1 5 10</td>
</tr>
</tbody>
</table>

For the MMKP, the instances set is summarized in Table 3.5. The header Instance, \( n \), \( \sum_{i} n_i \) and \( m_1 \) indicates respectively the name of the instance, the number of the groups, the number of the items of each group, the number of the total items, and the number of knapsack constraints. This instances set contains 13 instances (denoted \( I_{01}, \ldots, I_{13} \)) varying from small to large-scale size ones. These instances are given by Khan et al. (2002).

\textbf{Table 3.5: Test MMKP instances details}

<table>
<thead>
<tr>
<th>Instance</th>
<th>( n )</th>
<th>( \sum_{i} n_i )</th>
<th>( m_1 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>I01</td>
<td>5</td>
<td>5</td>
<td>25 5</td>
</tr>
<tr>
<td>I02</td>
<td>10</td>
<td>5</td>
<td>50 5</td>
</tr>
<tr>
<td>I03</td>
<td>15</td>
<td>10</td>
<td>150 10</td>
</tr>
<tr>
<td>I04</td>
<td>20</td>
<td>10</td>
<td>200 10</td>
</tr>
<tr>
<td>I05</td>
<td>25</td>
<td>10</td>
<td>250 10</td>
</tr>
<tr>
<td>I06</td>
<td>30</td>
<td>10</td>
<td>300 10</td>
</tr>
<tr>
<td>I07</td>
<td>100</td>
<td>10</td>
<td>1000 10</td>
</tr>
<tr>
<td>I08</td>
<td>150</td>
<td>10</td>
<td>1500 10</td>
</tr>
<tr>
<td>I09</td>
<td>200</td>
<td>10</td>
<td>2000 10</td>
</tr>
<tr>
<td>I10</td>
<td>250</td>
<td>10</td>
<td>2500 10</td>
</tr>
<tr>
<td>I11</td>
<td>300</td>
<td>10</td>
<td>3000 10</td>
</tr>
<tr>
<td>I12</td>
<td>350</td>
<td>10</td>
<td>3500 10</td>
</tr>
<tr>
<td>I13</td>
<td>400</td>
<td>10</td>
<td>4000 10</td>
</tr>
</tbody>
</table>

Because of the unavailability of the instances of MCKP and GUBMKP, we used the weakly correlated procedure proposed by Han et al. (2010a) to generate instances.
For the GUBMKP, we range the instances into four sets and we vary the number of items between 2500 and 15000. For each set, we generate 30 instances. In total we generate 120 instances. The instances sets are reported in Table 3.6 in which the headers Set, $n$, $n_i$, $m_1$ and $\sum n_i$ indicate the name of the sets, the number of the groups, the number of the items of each group, the number of the total items, and the number of knapsack constraints respectively.

Table 3.6: Test GUBMKP problem details

<table>
<thead>
<tr>
<th>Set</th>
<th>$n$</th>
<th>$n_i$</th>
<th>$\sum n_i$</th>
<th>$m_1$</th>
</tr>
</thead>
<tbody>
<tr>
<td>gubmkp1</td>
<td>10</td>
<td>250</td>
<td>2500</td>
<td>5</td>
</tr>
<tr>
<td>gubmkp2</td>
<td>10</td>
<td>500</td>
<td>5000</td>
<td>5</td>
</tr>
<tr>
<td>gubmkp3</td>
<td>10</td>
<td>1000</td>
<td>10000</td>
<td>5</td>
</tr>
<tr>
<td>gubmkp4</td>
<td>10</td>
<td>1500</td>
<td>15000</td>
<td>5</td>
</tr>
</tbody>
</table>

Table 3.7 shows the MCKP instances in which the headers Set, $n$, $n_i$ and $\sum n_i$ respectively indicates the name of the set, the number of the groups, the number of the items of each group, and the number of the total items. Note that each set $mckp$ for $i = (1, \ldots, 3)$ contains 30 instances.

Table 3.7: Test MCKP instances details

<table>
<thead>
<tr>
<th>Set</th>
<th>$n$</th>
<th>$n_i$</th>
<th>$\sum n_i$</th>
</tr>
</thead>
<tbody>
<tr>
<td>mckp1</td>
<td>1000</td>
<td>1000</td>
<td>1000000</td>
</tr>
<tr>
<td>mckp2</td>
<td>1500</td>
<td>1000</td>
<td>1500000</td>
</tr>
<tr>
<td>mckp3</td>
<td>2000</td>
<td>1000</td>
<td>2000000</td>
</tr>
</tbody>
</table>

3.5.2 Evaluation of the transformation

In this section, we give the results obtained by applying the different transformations between problems mentioned above. To compare the generated problems with the original ones, we use the percentage of deviation denoted $Dev$ which is calculated in the following way $Dev = \frac{T_2}{T_1} \times 100$. $T1$ and $T2$ represent respectively the CPU time value of the solution of the classical formulation denoted model1 and the CPU time value of the solution of the generated problem based on the
transformation denoted *model2*. First of all, all the results obtained validate the transformation between problems and reach the same values of optimality. Also the generated problems are able to give a reasonable computing time. We report, in Table 3.8, the overall results obtained by both formulations for all problems. The columns denoted $m_{T1}$ and $m_{T2}$ represent the average of computation time of *model1* and *model2* respectively, and *Dev* represents the percentage of deviation.

**Table 3.8: Performances comparison of the transformation between different ILPs**

<table>
<thead>
<tr>
<th>Set</th>
<th>$m_{T1}$</th>
<th>$m_{T2}$</th>
<th>Dev</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>GUBMKP→ MMKP</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>gubmkp1</td>
<td>1.8</td>
<td>1.3</td>
<td>71.2%</td>
</tr>
<tr>
<td>gubmkp2</td>
<td>2.8</td>
<td>2.3</td>
<td>82.6%</td>
</tr>
<tr>
<td>gubmkp3</td>
<td>5.0</td>
<td>4.3</td>
<td>84.7%</td>
</tr>
<tr>
<td>gubmkp4</td>
<td>8.4</td>
<td>8.2</td>
<td>98.0%</td>
</tr>
<tr>
<td><strong>MKP→ MMKP</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>mknapeb1</td>
<td>3.0</td>
<td>2.9</td>
<td>98.7%</td>
</tr>
<tr>
<td>mknapeb2</td>
<td>129.0</td>
<td>126.1</td>
<td>97.7%</td>
</tr>
<tr>
<td>mknapeb3</td>
<td>1743.8</td>
<td>1753.2</td>
<td>100.5%</td>
</tr>
<tr>
<td>mknapeb4</td>
<td>23.9</td>
<td>24.0</td>
<td>100.1%</td>
</tr>
<tr>
<td>mknapeb5</td>
<td>3088.2</td>
<td>3078.6</td>
<td>99.7%</td>
</tr>
<tr>
<td>mknapeb6</td>
<td>1952.1</td>
<td>1958.8</td>
<td>100.3%</td>
</tr>
<tr>
<td><strong>MDMKP→ MDMMKP</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>mdkmkp_ct1</td>
<td>48.3</td>
<td>48.2</td>
<td>99.9%</td>
</tr>
<tr>
<td>mdkmkp_ct2</td>
<td>2437.4</td>
<td>2431.9</td>
<td>99.8%</td>
</tr>
<tr>
<td>mdkmkp_ct3</td>
<td>2394.0</td>
<td>1939.1</td>
<td>81.0%</td>
</tr>
<tr>
<td>mdkmkp_ct4</td>
<td>1414.5</td>
<td>1408.3</td>
<td>99.6%</td>
</tr>
<tr>
<td>mdkmkp_ct5</td>
<td>2509.8</td>
<td>2501.5</td>
<td>99.7%</td>
</tr>
<tr>
<td>mdkmkp_ct6</td>
<td>3282.7</td>
<td>3238.4</td>
<td>98.6%</td>
</tr>
<tr>
<td><strong>MMKP→ MDMKP</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>l1</td>
<td>0</td>
<td>0</td>
<td>–</td>
</tr>
<tr>
<td>l2</td>
<td>0</td>
<td>0</td>
<td>–</td>
</tr>
<tr>
<td>l3</td>
<td>2</td>
<td>2</td>
<td>100.0%</td>
</tr>
<tr>
<td>l4</td>
<td>54</td>
<td>26</td>
<td>48.1%</td>
</tr>
<tr>
<td>l5</td>
<td>0</td>
<td>0</td>
<td>–</td>
</tr>
<tr>
<td>l6</td>
<td>0</td>
<td>0</td>
<td>–</td>
</tr>
<tr>
<td>l7</td>
<td>4771</td>
<td>4739</td>
<td>99.3%</td>
</tr>
<tr>
<td>l8</td>
<td>6052</td>
<td>4933</td>
<td>81.5%</td>
</tr>
<tr>
<td>l9</td>
<td>4735</td>
<td>4287</td>
<td>90.5%</td>
</tr>
<tr>
<td>l10</td>
<td>5411</td>
<td>4736</td>
<td>87.5%</td>
</tr>
<tr>
<td>l11</td>
<td>4517</td>
<td>4544</td>
<td>100.6%</td>
</tr>
<tr>
<td>l12</td>
<td>5494</td>
<td>4904</td>
<td>89.3%</td>
</tr>
<tr>
<td>l13</td>
<td>6700</td>
<td>5096</td>
<td>76.1%</td>
</tr>
<tr>
<td><strong>MCKP→ GUBMKP</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>mckp1</td>
<td>21.8</td>
<td>21.0</td>
<td>96.0%</td>
</tr>
<tr>
<td>mckp2</td>
<td>43.5</td>
<td>32.2</td>
<td>73.9%</td>
</tr>
<tr>
<td>mckp3</td>
<td>44.8</td>
<td>43.5</td>
<td>97.1%</td>
</tr>
</tbody>
</table>
For the transformation of the GUBMKP into the MMKP, we notice that the generated formulations (model2) use less time than the original ones (model1) for all sets and the average of the Dev is equal to 84.1%.

Also, we notice the same for the transformation of the MDKP into the MDMMKP and the transformation of the MCKP into the GUBMKP who values of the average of the Dev are equal to 96.4% and 89% respectively.

For the transformation of the MKP into the MMKP, we notice that all the values reached by the two formulations are very close and the average of the deviation of CPU time value Dev is equal to 99.5%.

For the transformation of the MMKP into the MDMKP, we remark that the generated formulations (model2) use less time than the original ones model1 except the instance I11. We note that the average of the Dev for the whole instances is equal to 85.9%.

We do not test the transformation of the GUBMKP into the MKP because the similarity of the lp file of CPLEX for both formulations as CPLEX does not show the zero multipliers in the lp file.

### 3.6 Conclusion

This chapter presented a set of knapsack problems involving dimensions, demands and multiple choice constraints among which are the MKP, the MDMKP, the MCKP, the MMKP and the GUBMKP. Specifically, we defined the multiple demand multidimensional multiple choice knapsack problem (MDMMKP) as a generalization of these problems. Moreover, we applied a set of transformations between the different integer linear programs of knapsack extensions.
Using these transformations, we showed that any algorithm able to solve the generalized problem can definitely solve its related problems mentioned above. Computational results indicate that solving the new formulations using the transformations is able to generate reasonable computing time compared with the original ones.
Chapter 4

The multiple choice multidimensional knapsack constraint

4.1 Introduction

Global constraints are one of the distinguishing features of constraint programming. See, for example, Bessière et al. (2006), Régin (1999, 1994), Pesant (2004). They specify patterns that occur in many problems, and exploit efficient and effective propagation algorithms to prune search space. Different categories of global constraints are identified Régin (2003). In this chapter, we will consider a constraint belonging to the weighted constraints. This category contains constraints which are associated with some costs, usually a summation is implied and there is a limit on it. Among these constraints is: the knapsack constraint.

In the literature, knapsack constraint refers to two problems. The two sided knapsack constraint proposed by Trick Trick (2003b), under the name ”knapsack constraint”, which is considered as subset-sum constraint. Trick proposed a pseudo polynomial filtering algorithm based on dynamic programming approach to solve it. On the other hand, Meinolf Sellmann (2003) introduced a knapsack constraint
involving an objective to maximize. This constraint is closer to the classical knapsack problem $kmax$ Kellerer et al. (2004). Meinolf Sellmann (2003) introduced the theoretical concept of approximated consistency. The main idea of the algorithm consists in using bounds of guaranteed accuracy for cost-based filtering of optimization constraints. It was shown theoretically that the proposed algorithm achieves approximated consistency in amortized linear time for bounds with arbitrary but constant accuracy. Also, Meinolf Sellmann (2004) provided an empirical evaluation of approximated consistency for knapsack constraints by applying it on the market split problem and the automatic recording problem. Experiments results showed that approximated consistency for knapsack constraints can lead to massive improvements for critically constrained problems.

In this chapter, we aim to introduce a new weighted constraint: the *multiple choice multidimensional knapsack constraint* $mcmdk$. This constraint can be viewed as a special case of the classical knapsack problem where resource is multidimensional and items are divided on sets. Note that no optimization criteria is taken into account. $mcmdk$ constraints are common to almost real-life applications. For instance, they are present in the resource allocation, the management of resources in multimedia systems and in telecommunication networks.

The rest of the chapter is organized as follows. Section 2 gives the necessary preliminaries of constraint programming. Section 3 discusses and formulates the $mcmdk$ constraint by conjunctions of $sum$ and $implies$ constraints. In section 4, we propose a filtering algorithm for propagating this global constraint while section 5 evaluates the algorithm experimentally. Finally, Section 6 summarizes the contributions of this work.

### 4.2 Constraint programming preliminaries

This section provides basic constraint programming concepts and present two constraints of interest: $sum$ and $implies$. For further information on constraint programming we refer the reader to Rossi et al. (2006a).
4.2.1 Constraint programming

The *domain* of a variable $x$, denoted by $D(x)$, is a set of ordered values that can be assigned to $x$. Let $X = x_1, x_2, \ldots, x_n$ be a sequence of variables. The minimum (resp. maximum) value of the domain is denoted by $\min(D(x))$ (resp. $\max(D(x))$).

A *constraint* $C$ on $X$ is defined as a subset of the Cartesian product of the domains of the variables in $X$: $C \subseteq D(x_1) \times D(x_2) \times \ldots, \times D(x_n)$. A tuple $(d_1, \ldots, d_n) \in C$ is called a solution to $C$. We also say that the tuple satisfies $C$. A value $d \in D(x_i)$ for some $i = 1, \ldots, n$ is *inconsistent* with respect to $C$ if it does not belong to a tuple of $C$, otherwise it is *consistent*. $C$ is *inconsistent* if it does not contain a solution. Otherwise, $C$ is called *consistent*.

A *Constraint Satisfaction Problem* (CSP) is defined by a finite sequence of variables $X = x_1, \ldots, x_n$, together with a finite set of constraint $C$, each on a subset of $X$. A *solution* to a CSP is an assignment of variables to values in their respective domains such that all of the constraints are satisfied.

*Constraint Programming* proposes to solve CSPs by associating with each constraint a *filtering algorithm* that removes some domain values which cannot belong to any solution of the CSP. These filtering algorithms are repeatedly called until no more domain value can be removed. Then, CP uses a search procedure where filtering algorithms are systematically applied when the variable domain is modified. Therefore, with respect to the current domains of the variables and thanks to filtering algorithms, CP removes, once and for all, certain inconsistencies that would have been discovered several times otherwise.

In order to be effective, filtering algorithms should be efficient, because they are applied many times during the solution process. Further, they should remove as many inconsistent values as possible. If a filtering algorithm for a constraint $C$ removes all inconsistent values from the domains with respect to $C$, we say that it establishes the *arc consistency* of $C$. Arc consistency is also referred as *domain consistency* or to *generalized-arc consistency* (GAC).
4.2.2 sum and implies constraints

**sum constraint:** The sum constraint is one of the most frequently occurring constraints in applications. Let \( x_1, \ldots, x_n \) and \( y \) be variables. The sum constraint is defined as follows: \( y = \sum_{i=1}^{n} x_i \)

**implies constraint:** Let \( c_1 \) and \( c_2 \) be constraints. The implies constraint \( \text{implies} (c_1, c_2) \) states that if \( c_1 \) holds then \( c_2 \) holds where \( c_1 \) represents the condition constraint and \( c_2 \) the conclusion constraint. We also write \( c_1 \Rightarrow c_2 \).

4.3 The multiple choice multidimensional knapsack constraint

The multiple choice multidimensional knapsack constraint is defined by a set of items divided on \( n \) disjointed sets \( S = (S_1 \cup S_2 \cup \cdots \cup S_n) \) and a resource with \( m \) dimensions. The resource has a limited capacity in each dimension \( k \) denoted by \( c_k \). Each item \( j \) belonging to a set \( S_i \) has a weight in each dimension, denoted by \( w_{k,i,j} \). The objective is to select exactly one item from each set so that their overall weight does not exceed any resource capacity.

Such multiple choice multidimensional knapsack constraint \( mcmdk \) can be achieved in the following way:

We introduce for each set \( S_i \) these variables (i) \( x_i \) whose value corresponds to the index of the selected item of this set and (ii) \( y_{k,i} \) whose value corresponds to the weight of the selected item for the dimension \( k \). Using these variables combined with implies and sum constraints, the \( mcmdk \) constraint can be formulated as follows:

\[
mcmdk \begin{cases} 
\sum_{i=1}^{n} y_{k,i} \leq c_k & \forall k \in 1, \ldots, m \\
(x_i = j) \Rightarrow (y_{k,i} = w_{k,i,j} \forall k \in 1, \ldots, m) & \forall i \in 1, \ldots, n, \forall j \in 1, \ldots, |S_i| \\
x_i \in \{1, \ldots, |S_i|\} & \forall i \in 1, \ldots, n \\
y_{k,i} = \{w_{k,i,j}|\forall j \in 1, \ldots, |S_i|\} & \forall i \in 1, \ldots, n, \forall k \in 1, \ldots, m
\end{cases}
\]
Chapter 3. The multiple choice multidimensional knapsack constraint

The first set of constraints (equation (1)) calculates the weight of the selected item in each dimension and checks that the capacity in each dimension is not exceeded. The second set of constraints (equation (2)) linked variables $y$ with variables $x$ using the implies constraint.

**Definition 1:** More formally, the \textit{mcmdk} constraint can be defined as follows:

Let $x_1, \ldots, x_n, y_{1,1}, \ldots, y_{m,1}, y_{1,2}, \ldots, y_{m,2}, \ldots, y_{1,n}, \ldots, y_{m,n}$ be variables with respective finite domains $D(x_1), \ldots, D(x_n), D(y_{1,1}), \ldots, D(y_{m,1}), D(y_{1,2}), \ldots, D(y_{m,2}), \ldots, D(y_{1,n}), \ldots, D(y_{m,n})$, let $w_{k,i,j} \in \mathbb{Q}$ for $k = 1, \ldots, m$, $i = 1, \ldots, n$ and all $j \in \bigcup_{i=1,\ldots,n} D(x_i)$ be integers and let $c$ be an array of integers, i.e. $c = [c_1, c_2, \ldots, c_m]$. Then, the \textit{mcmdk} constraint is defined as:

$$mcmdk([x_1, \ldots, x_n], [y_{1,1}, \ldots, y_{m,1}, y_{1,2}, \ldots, y_{m,2}, \ldots, y_{1,n}, \ldots, y_{m,n}], w, c) \text{ iff }$$

$$\forall k \in 1, \ldots, m \sum_{i=1}^n y_{k,i} \leq c_k \land \forall i \in 1, \ldots, n, \forall j \in D(x_i), (x_i = j) \Rightarrow (y_{k,i} = w_{k,i,j} \forall k \in 1, \ldots, m)$$

**Example 1:** There follows (Fig 1) a small illustrative example which will be used throughout the chapter. We consider 3 sets each which is composed by 4 items and a resource with 2 dimensions. Dimensions capacities are $c = [14, 14]$ and weights of items are the following:

$$w = \begin{bmatrix} [[4, 2, 9, 11], [3, 11, 0, 5], [7, 1, 6, 10]], \\
[[2, 4, 3, 7], [1, 0, 9, 6], [8, 13, 10, 5]] \end{bmatrix}$$

### 4.3.1 Fundamental properties

Some classical properties can be applied in the pre-processing phase of the filtering algorithm of \textit{mcmdk} constraint:

1. If there exists $k \in 1, \ldots, m$ such that $\sum_{i=1}^n min(D(y_{k,i})) > c_k$ then \textit{mcmdk} is inconsistent.
Chapter 3. The multiple choice multidimensional knapsack constraint

Input:
\[ n \leftarrow 3 \quad m \leftarrow 2 \]
\[ \rightarrow \text{Weights} \rightarrow \]
\[
\begin{array}{cccccccc}
w_{1,i,j} & 4 & 2 & 9 & 11 & 3 & 11 & 0 & 5 & 7 & 1 & 6 & 10 \\
w_{2,i,j} & 2 & 4 & 3 & 7 & 1 & 0 & 9 & 6 & 8 & 13 & 10 & 5 \\
\end{array}
\]
\[ \rightarrow \text{Capacities} \rightarrow \]
\[ c_1 \leftarrow 14 \quad c_2 \leftarrow 14 \]

Constraints:
\[ y_{1,1} + y_{1,2} + y_{1,3} \leq 14 \]
\[ y_{2,1} + y_{2,2} + y_{2,3} \leq 14 \]
\[ x_1 = j \ (j = 1, \ldots, 4) \Rightarrow y_{1,1} = w_{1,1,j}, \ y_{2,1} = w_{2,1,j} \]
\[ x_2 = j \ (j = 1, \ldots, 4) \Rightarrow y_{2,1} = w_{2,1,j}, \ y_{2,2} = w_{2,2,j} \]
\[ x_3 = j \ (j = 1, \ldots, 4) \Rightarrow y_{3,1} = w_{3,1,j}, \ y_{3,2} = w_{3,2,j} \]

Variables:
\[ y_{1,1} = \{2, 4, 9, 11\} \quad y_{1,2} = \{0, 3, 5, 11\} \quad y_{1,3} = \{1, 6, 7, 10\} \]
\[ y_{2,1} = \{2, 3, 4, 7\} \quad y_{2,2} = \{0, 1, 6, 9\} \quad y_{2,3} = \{5, 8, 10, 13\} \]
\[ x_1 = \{1, 2, 3, 4\} \quad x_2 = \{1, 2, 3, 4\} \quad x_3 = \{1, 2, 3, 4\} \]

Figure 4.1: An instance example of the multiple choice multidimensional knapsack constraint

2. If there exists \( k \in 1, \ldots, m \) such that \( w_{k,i,j} > c_k (i \in 1, \ldots, n, \ j \in D(x_i)) \)
then the \( w_{k,i,j} \) is inconsistent.

3. If there exists \( k \in 1, \ldots, m \) such that \( \sum_{i=1}^{n} \max(D(y_{k,i})) \leq c^k \) then the \( k \)-th resource dimension can be eliminate.

Property 4: \( \forall f \in 1, \ldots, n \ \forall k \in 1, \ldots, m \), let \( lb = c_k - \sum_{i=1,i \neq f}^{n} \min(D(y_{k,i})) \)
then each value \( v \) of \( D(y_{k,f}) \geq lb \) is inconsistent and values \( j \) of \( D(x_f) \), \( w_{k,f,j} \)
of \( D(y_{k,f}) \ \forall k = (1, \ldots, m) \) are also inconsistent, where \( j \) is the value that corresponds to item \( j \) of set \( S_f \) whose weight \( w_{k,f,j} \) is equal to \( v \)

Proof: To be consistent, the \( mcmdk \) constraint must check the following inequation:

\[
\sum_{i=1}^{n} y_{k,i} \leq c_k
\]

Thus, for a given \( f \in 1, \ldots, n \)

\[
c_k - \sum_{i=1,i \neq f}^{n} y_{k,i} \geq y_{k,f}
\]
The search for a support for a value \( v \) of \( D(y_{k,f}) \) is immediate because any value \( b \) of \( \sum_{i=1,i\neq f}^{n} D(y_{k,i}) \) such that \( c_k - b \geq v \) is a support, so \( v \) is consistent with the constraint if

\[
v \geq \min(c_k - \sum_{i=1,i\neq f}^{n} D(y_{k,i}))
\]

We can immediately state that all values of \( D(y_{k,f}) \) less than or equal to

\[
c_k - \sum_{i=1,i\neq f}^{n} \min(D(y_{k,i}))
\]

can be removed.

Note that, by definition of the problem, each item of \( S_f \) is presented by a value in each \( y_{k,f} \) variables \( k \in 1, \ldots, m \) and by a value in \( x_f \) variable.

Now, let \( j \) the value of \( x_f \) such that \( w_{k,f,j} \) is equal to \( v \). When \( v \) of \( D(y_{k,f}) \) is inconsistent, then their correspondent values \( w_{k,f,j} \) of \( D(y_{k,f}) \), \( k \in 1, \ldots, m \) and \( j \) of \( D(x_f) \) are also inconsistent (the item \( j \) of \( S_f \) cannot configure on any feasible solution).

**Proposition:** Finding a solution to \( mcmdk \) constraint is \( NP \)-complete problem in general.

**Proof:** This problem is obviously in \( NP \) (easy polynomial certificate). The \( mcmdk \) constraint is an extension of the multiple choice knapsack problem \( mckp \) Kellerer et al. (2004). The \( mckp \) is \( NP \)-Hard Kellerer et al. (2004) as it contains \( kp \) as special case. Note that no optimization criteria is taken into account, then the \( mcmdk \) is \( NP \)-complete.
4.4 Filtering algorithm for \textit{mcmdk} constraint

The key idea behind the algorithm is based on property 4 which removes values from \( y \) variables, and propagates the consequences of this removal. In fact, the removing of a value of \( D(y_{k,f}) \), \( k \in 1,\ldots,m \) involves the deletions of its correspondent values of \( D(x_f) \) and of \( D(y_{k_2,f}) \), \( k_2 \in 1,\ldots,m \) \( k_2 \neq k \).

Since the removing of values of \( D(y_{k,f})f = (1,\ldots,n) \), \( k = (1,\ldots,m) \) depends on \( \min(D(y_{k,f})) \), \( f_2 \in 1,\ldots,n \) \( f_2 \neq f \), the proposed filtering algorithm must be called only when a \( \min(D(y_{k,f})) \) are modified. It is useless to call it for other modification.

\begin{verbatim}
begin
for k ← 1 to m do
   for i ← 1 to n do
      put(i, k, Q);
   end
end
\end{verbatim}

\textbf{Procedure} initialize(Q,m,n:Integer)

The main filtering algorithm (see Algorithm 1) has two phases: initialization of a list \( Q \) (line 2 calls procedure \textbf{initialize}) and propagation (lines 3-20).

Procedure \textbf{initialize} is committed to initializing the list \( Q \) (\( Q \) is a list that stores pairs \( \langle \text{knapsack}, \text{set} \rangle \) awaiting further processing).

Once procedure \textbf{initialize} has finished, algorithm 1 begins the propagation phase. This process propagates the consequences of the removal of values from \( y \) variables. Thus, the pair \( \langle k, f \rangle \) stored in \( Q \) is selected (line 4) and its corresponding \( y_{k,f} \) variable is revised in the following.

At line 5, the function \textit{getInf}(\textit{Variable} \( y[][] \), \textit{Integer} \( k \), \textit{Integer} \( f \)) returns the sum of the minimum values of variables \( y_{k,i} \forall i \neq f i = (1,\ldots,n) \).

Next, at line 6, if the sum of the value \textit{sum} added to value \textit{val} which corresponds to the maximum value of the variable \( y_{k,f} \) (value returned by the function
begin
Initialize($Q, n, m$);
while $Q! = \emptyset$ do
    $\langle k, f \rangle \leftarrow get(Q)$;
    $sum \leftarrow getInf(y, k, f)$;
    while $sum + getSup(y[k][f]) > c[k]$ do
        $val \leftarrow getSup(y[k][f])$;
        $remVal(y[k][f], val)$;
        $j \leftarrow getIndex(k, f, val, y)$;
        $remVal(x_f, j)$;
        for $k2 \leftarrow 1$ to $m$ do
            if $k2 \neq k$ then
                if $w[k2][f][j] = getInf(y[k2][f])$ then
                    $putSet(f2, f, k2, Q)$;
                    $remVal(y[k2][f], w[k2][f][j])$;
                end
            end
        end
    end
end

Algorithm 1: Filtering algorithm

$getSup(Variable\ y)$ exceeded the capacity $c[k]$, then the value $val$ is considered inconsistent and it will be removed. Thus, we propagate the consequences of the removal of $val$ (lines 9 - 18).

Let $j$ the value (returned by the function $getIndex$) corresponds to item $j$ of set $S_f$ whose weight $w_{k,f,j}$ is equal to $val$. The removal of the item $j$ from the set $S_i$ is done by removing:

- the value $j$ of $D(x_f)$ (line 10).
- the value $w_{k2,f,j}$ of $D(y_{k2,f})$ $k2 \in 1, \ldots, m$, $k2 \neq k$. (line 16).

Every time that it is detected that a lower value of a variable $y_{k2,f}$ is removed, the pairs $\langle k2, f2 \rangle \forall k2 = (1, \ldots, m)\ k2 \neq k,\ f2 = (1, \ldots, n)\ \forall f2 \neq f$ is added to the list $Q$ for subsequent propagation using the procedure $putSet$. (line 14)

Next, we return to line 3, we select the next pair $\langle k, f \rangle$ and we repeat the same process until the list $Q$ becomes empty.
Table 4.1: A worked example

<table>
<thead>
<tr>
<th>pair</th>
<th>satisfaction</th>
<th>variables</th>
<th>list Q</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1, 1)</td>
<td>(0 + 1) + 11 ≤ 14</td>
<td>–</td>
<td>(2, 3) ⟨2, 2⟩ ⟨2, 1⟩ ⟨1, 3⟩ ⟨1, 1⟩</td>
</tr>
<tr>
<td>(1, 2)</td>
<td>(1 + 2) + 11 ≤ 14</td>
<td>–</td>
<td>(2, 3) ⟨2, 2⟩ ⟨2, 1⟩ ⟨1, 3⟩</td>
</tr>
<tr>
<td>(1, 3)</td>
<td>(2 + 0) + 10 ≤ 14</td>
<td>–</td>
<td>(2, 3) ⟨2, 2⟩ ⟨2, 1⟩</td>
</tr>
<tr>
<td>(2, 1)</td>
<td>(0 + 5) + 7 ≤ 14</td>
<td>–</td>
<td>(2, 3) ⟨2, 2⟩</td>
</tr>
<tr>
<td>(2, 2)</td>
<td>(2 + 5) + 9 &gt; 14</td>
<td>(y_{2,2} = {0, 1, 6, 9}) (y_{1,2} = {6, 3, 5, 11})</td>
<td>(1, 3) ⟨1, 1⟩ ⟨2, 3⟩</td>
</tr>
<tr>
<td>(2, 3)</td>
<td>(2 + 0) + 13 &gt; 14</td>
<td>(y_{2,3} = {5, 8, 10, 13}) (y_{1,3} = {1, 6, 7, 10})</td>
<td>(1, 2) ⟨1, 3⟩, ⟨1, 1⟩</td>
</tr>
<tr>
<td>(1, 1)</td>
<td>(3 + 6) + 11 &gt; 14</td>
<td>(y_{1,1} = {2, 4, 9, 14}) (y_{2,1} = {2, 3, 4, 7})</td>
<td>(1, 2) ⟨1, 3⟩</td>
</tr>
<tr>
<td></td>
<td>(3 + 6) + 9 &gt; 14</td>
<td>(y_{1,1} = {2, 4, 9}) (y_{2,1} = {2, 3, 4})</td>
<td></td>
</tr>
<tr>
<td>(1, 3)</td>
<td>(2 + 3) + 10 &gt; 14</td>
<td>(y_{1,3} = {6, 7, 10}) (y_{2,3} = {5, 8, 10})</td>
<td>(2, 2), ⟨2, 1⟩, ⟨1, 2⟩</td>
</tr>
<tr>
<td>(1, 2)</td>
<td>(2 + 6) + 11 &gt; 14</td>
<td>(y_{1,2} = {3, 5, 11}) (y_{2,2} = {6, 1, 6})</td>
<td>(2, 3), ⟨2, 2⟩, ⟨2, 1⟩</td>
</tr>
<tr>
<td>(2, 1)</td>
<td>(1 + 8) + 4 ≤ 14</td>
<td>–</td>
<td>(2, 3), ⟨2, 2⟩</td>
</tr>
<tr>
<td>(2, 2)</td>
<td>(2 + 8) + 6 &gt; 14</td>
<td>(y_{2,2} = {1, 6}) (y_{1,2} = {3, 5})</td>
<td>(2, 3)</td>
</tr>
<tr>
<td>(2, 3)</td>
<td>(2 + 1) + 10 ≤ 14</td>
<td>–</td>
<td></td>
</tr>
</tbody>
</table>

4.4.1 A worked example

Table 4.1 highlight the proposed algorithm by considering the initial example.

In a first phase, which ids initialization and is executed only once, the algorithm initializes the list \(Q\): \(Q = \{(2, 3) \langle 2, 2 \rangle \langle 2, 1 \rangle \langle 1, 3 \rangle \langle 1, 2 \rangle \langle 1, 1 \rangle \}\).

Each line of the table 4.1 represents the work done after the selection of a pair \(\langle k, f \rangle\) stored in \(Q\) (line 4-19 Algorithm 1). The first column \(pair\) represents the selected pair stored in \(Q\) (line 4 Algorithm 1). Column 2 \(satisfaction\) represents the satisfaction test of resource capacity \(k\) (line 6 Algorithm 1). Column 3 \(variables\) represents the domain modification of \(y\) variables. The column \(list Q\) represents the list \(Q\).

**Theorem:** The filtering algorithm does not maintain generalized arc consistency.  
**Proof:** Generalized arc consistency implies that every value is consistent. To show the reverse, we present an example in which our filtering algorithm cannot prune all inconsistent values.

We consider an example with 2 sets of items each one of these sets is composed by 3 items and a resource with 2 dimensions. The first set \(set_1\) is composed by \(set_1 = \{(2, 4), (6, 1), (7, 3)\}\) and the second set \(set_2\) is composed by \(set_2 = \)
Let \( c = \{ c_1 = 8, c_2 = 6 \} \). Then the \( y \) variables are defined as follows: \( y_{1,1} = \{2, 6, 7\} \), \( y_{1,2} = \{1, 5, 6\} \), \( y_{2,1} = \{1, 3, 4\} \) and \( y_{2,2} = \{2, 4, 5\} \).

Applying the proposed filtering algorithm to this example, no values are removed. Nevertheless, both items (7, 3) of \( S_1 \) and (6, 4) of \( S_2 \) cannot configure on any feasible solution, i.e. values 7 of \( D(y_{1,1}) \) 6 of \( D(y_{1,2}) \) 3 of \( D(y_{2,1}) \) and 4 of \( D(y_{2,2}) \) are inconsistent. Therefore, our filtering algorithm is not GAC.

### 4.5 Experiments

The purpose of our computational experiments to show that propagating \( mcmdk \) using the algorithm introduced in this chapter is more effective and more efficient than propagating it using the straightforward conjunction:

\[
\begin{align*}
mcmdk([x_1, \ldots, x_n], [y_{1,1}, \ldots, y_{m,1}, y_{1,2}, \ldots, y_{m,2}, \ldots, y_{1,n}, \ldots, y_{m,n}], w, c) \text{ iff } \\
\forall k \in 1, \ldots, m \quad \sum_{i=1}^{n} y_{k,i} \leq c_k \\
\forall i \in 1, \ldots, n, \forall j \in D(x_i), \quad (x_i = j) \Rightarrow (y_{k,i} = w_{k,i,j} \forall k \in 1, \ldots, m)
\end{align*}
\]

We name our filtering algorithm \textbf{mcmdk-fal} and the straightforward conjunction \textbf{mcmdk-conj}

To evaluate the filtering algorithm we propose randomly instances which can be described with three parameters, the number of sets \( n \), the number of items for each set \( |S_i| \) and the number of resource dimensions \( m \). These instances are generated in the following manner:

We classify the proposed instances in two classes. The first class is described by a fixed number of sets \((n = 15)\) and items in each set \((|S_i| = 15)\) and varying the number of dimensions \( m = 10, 12, \ldots, 28 \). The second class of instances is described by a fixed \((m = 15)\) and items \((|S_i| = 15)\) and a growing number of sets \( n = 10, 12, \ldots, 38 \). The weighed values are selected randomly between 0 and \( R = 100 \) and the resource capacity \( c_k \) is calculated as follows:
Chapter 3. The multiple choice multidimensional knapsack constraint

\[ c_k = \frac{1}{2} \left( \sum_{i=1}^{n} \min_{1 \leq j \leq |S_i|} w_{k,i,j} + \sum_{i=1}^{n} \max_{1 \leq j \leq |S_i|} w_{k,i,j} \right) \]

Ten instances were generated randomly for each combination of parameters, for a total 250 instances.

We implemented our filtering algorithm using the Java-based constraint programming engine Choco 2.1.5. The hardware used for the experiments is a 2.66 Ghz Intel Core 2 DUO processor with 3 Gb RAM running Windows XP.

The results are shown in table 4.2 and table 4.3. Table 4.2 reports on instances of class 1 with a fixed number of sets \( n = 15 \) and varying \( m \). Table 2 reports on instances of class 2 with a fixe \( m = 15 \) and varying \( n \). The runtimes in seconds (CPU), the number of backtracks (BT) and the number of visiting nodes (NODE) are reported as the average over instances with the same parameters.

<table>
<thead>
<tr>
<th>( n )</th>
<th>( m )</th>
<th>( \text{mcmdk-conj} )</th>
<th>( \text{mcmdk-fal} )</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>CPU</td>
<td>BT</td>
<td>NODE</td>
</tr>
<tr>
<td>15</td>
<td>10</td>
<td>0</td>
<td>25</td>
</tr>
<tr>
<td>12</td>
<td>1</td>
<td>154</td>
<td>700</td>
</tr>
<tr>
<td>14</td>
<td>22</td>
<td>2121</td>
<td>10456</td>
</tr>
<tr>
<td>16</td>
<td>55</td>
<td>4565</td>
<td>23044</td>
</tr>
<tr>
<td>18</td>
<td>127</td>
<td>9042</td>
<td>46601</td>
</tr>
<tr>
<td>20</td>
<td>1051</td>
<td>64150</td>
<td>340038</td>
</tr>
<tr>
<td>22</td>
<td>818</td>
<td>45101</td>
<td>236492</td>
</tr>
<tr>
<td>24</td>
<td>1218</td>
<td>61068</td>
<td>328298</td>
</tr>
<tr>
<td>26</td>
<td>977</td>
<td>49601</td>
<td>264322</td>
</tr>
<tr>
<td>28</td>
<td>1729</td>
<td>80489</td>
<td>427152</td>
</tr>
</tbody>
</table>

Table 4.2: Comparison on instances with \( n = 15 \)

As predicted by the complexity of \textit{mcmdk}, \textit{mcmdk-fal} and \textit{mcmdk-conj} become more time consuming as \( m \) grows in the first table and \( n \) grows in the second table. Tables 2 and 3 clearly show the impact of \textit{mcmdk-fal}. We observe that our proposed \textit{mcmdk-fal} introduced in this chapter performs way better than the basic model in all tested instances.
Chapter 3. The multiple choice multidimensional knapsack constraint

<table>
<thead>
<tr>
<th>n</th>
<th>m</th>
<th>mcmdk-conj</th>
<th>mcmdk-fal</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>CPU</td>
<td>BT</td>
</tr>
<tr>
<td>10</td>
<td>15</td>
<td>4</td>
<td>410</td>
</tr>
<tr>
<td>12</td>
<td>15</td>
<td>78</td>
<td>7228</td>
</tr>
<tr>
<td>14</td>
<td>15</td>
<td>45</td>
<td>3921</td>
</tr>
<tr>
<td>16</td>
<td>15</td>
<td>132</td>
<td>11175</td>
</tr>
<tr>
<td>18</td>
<td>15</td>
<td>139</td>
<td>11415</td>
</tr>
<tr>
<td>20</td>
<td>15</td>
<td>117</td>
<td>9399</td>
</tr>
<tr>
<td>22</td>
<td>15</td>
<td>121</td>
<td>10747</td>
</tr>
<tr>
<td>24</td>
<td>15</td>
<td>222</td>
<td>16632</td>
</tr>
<tr>
<td>26</td>
<td>15</td>
<td>248</td>
<td>18736</td>
</tr>
<tr>
<td>28</td>
<td>15</td>
<td>416</td>
<td>28751</td>
</tr>
<tr>
<td>30</td>
<td>15</td>
<td>211</td>
<td>15162</td>
</tr>
<tr>
<td>32</td>
<td>15</td>
<td>391</td>
<td>30275</td>
</tr>
<tr>
<td>34</td>
<td>15</td>
<td>1129</td>
<td>85582</td>
</tr>
<tr>
<td>36</td>
<td>15</td>
<td>997</td>
<td>61378</td>
</tr>
<tr>
<td>38</td>
<td>15</td>
<td>979</td>
<td>64656</td>
</tr>
</tbody>
</table>

Table 4.3: Comparison on instances with $m = 15$

4.6 Conclusion

In this chapter we have presented \textit{mcmdk}, a new global constraint belonging to the weighted constraints and closely related to the knapsack constraint. We have shown how the \textit{mcmdk} constraint can be modelled using the conjunction of \textit{sum} and \textit{implies} constraints. We have shown that the \textit{mcmdk} constrain is \textit{NP}-complete. Also, we have proposed a simple filtering algorithm to solve it. Our experiments show that propagating the \textit{mcmdk} constraint via the proposed filtering algorithm is more effective and efficient than propagating it using the straightforward conjunction.
Chapter 5

Solving Constrained Optimization Problems By Solution-based Decomposition Search

5.1 Introduction

Constrained optimization problems (COPs) are a generalization of the classical Constraint Satisfaction Problems (CSPs) with an associated function that must be optimized. COPs appear in many theoretical problems as well as for many real-life applications. Examples are production planning subject to demand and resource availability so that profit is maximized, air traffic control subject to safety protocols so that flight times are minimized and the transportation vehicles so that delivery times and fuel expenses are minimized.

Solving COPs is a challenging task and these problems are NP-Hard in general. One of the most common approaches to solve COPs is the Branch and Bound algorithm ($B&B$) which is a systematic enumeration method of all candidate solutions, where large subsets of sub-optimal candidate solutions are pruned by using upper (or lower) bounds of the objective function.
In this chapter we present a new strategy for solving COPs called solve and decompose (or S&D for short). The proposed strategy is a systematic iterative depth-first strategy that is based on problem decomposition. At the root node the corresponding CSP (of the COP) is solved and a feasible solution is computed. Unlike other decomposition methods, S&D uses a feasible solution of the CSP of the original COP to further decompose the original problem into a bounded number of subproblems which are considerably smaller in size. It also uses the value of the feasible solution as a bound that we add to the created subproblems in order to strengthen the cost-based filtering. Furthermore, the feasible solution is exploited in order to create subproblems that have more promise in finding better solutions which are explored in a depth-first manner. The whole process is repeated until we reach a specified depth where we do not decompose the subproblems anymore but we solve them to optimality using B&B (or any other exact method). Our initial results on two benchmark problems show that S&D may reach improvement of up to three orders of magnitude in terms of runtime when compared to B&B.

The remainder of this chapter is organized as follows. Section 5.2 provides the necessary formal background. We introduce a basic version of S&D in Section 5.3 and improve it in Section 5.4. Computational results are given in Section 5.5 whereas the related work is described in Section 5.6. Finally, section 5.7 concludes this chapter.

5.2 Formal background

The Constraint Satisfaction Problem (CSP) Tsang (1993), Rossi et al. (2006b) offers a powerful framework for representing and efficiently solving many NP-hard problems. A CSP is defined by a triplet \((X, D, C)\) where \(X\) is a set of decision variables \(X = \{x_1, \ldots, x_n\}\), \(D\) is a set of domain values associated with \(x_1, \ldots, x_n\) respectively and \(C\) is set of constraints. The domain of \(x_i\) is referred to as \(D(x_i)\). Each constraint is a relation specifying allowed combinations of values for some
variables. A solution to a CSP is an assignment of variables to values in their respective domains such that all of the constraints are satisfied.

In many applications, some solutions are better than others. The task in such problems is to find optimal solutions, where optimality is defined in terms of some application-specific functions. These problems are called Constrained Optimization Problems (COPs) Jain and Grossmann (2001). A COP is defined by 4-tuple \((X, D, C, f)\) where \((X, D, C)\) is a CSP and \(f\) is an objective function that maps every solution to a numerical value. The objective of COP is to find a feasible solution such that its value is either maximized or minimized, depending on the requirements of the problem. Throughout this chapter, the set of variables \(X\) is partitioned into two types of variables \(Y\) and \(Z\) such that \(X = Y \cup Z\). The set of variables \(Y\) represents the variables appearing in the objective function (and the problem constraints) whereas \(Z\) is the set of variables appearing only in the problem constraints (but not in the objective function).

One of the most common approaches for finding optimal solutions is the Branch and Bound algorithm (\(B&B\)) Lawler and Wood (1966). \(B&B\) is a backtracking algorithm storing the value of the best solution found during search and uses it to prune parts of the search tree. More precisely, whenever \(B&B\) encounters a partial solution that cannot be extended to form a solution of better value, the algorithm backtracks.

### 5.3 The basic Solve and Decompose algorithm

In this section, we describe a basic version of our algorithm called Solve and Decompose and refer to it as \(S&D\) in short.

\(S&D\) is a solution-based decomposition. The feasible solution is obtained by solving the CSP of the original COP. The feasible solution is then exploited in order to (i) decompose the original problem into a set of subproblems which are considerably smaller in size than the original problem; (ii) identify promising regions
of the search tree; and (iii) strengthen the cost-based filtering during the solving process.

5.3.1 The decomposition method

Consider a minimization COP \((X, D, C, f)\) where \(X = Y \cup Z\) and a solution \(S\) to the CSP \((X, D, C)\). We denote by \(\bar{x}_i\) the value assigned to \(x_i\) in \(S\). For each variable \(y_i \in Y\), we create two sets of values: \(\text{left}_i\) which contains all values in the domain of \(y_i\) less than or equal to \(\bar{y}_i\) and \(\text{right}_i\) which contains all values strictly greater than \(\bar{y}_i\). Thus, \(\text{left}_i\) and \(\text{right}_i\) form a partition of the domain of \(y_i\).

Using \(\text{left}_i\), \(\text{right}_i\), there exists \(2^{|Y|}\) possible subproblems of \((X, D, C)\). Each subproblem is created by modifying the domains of the \(y_i\)'s as follows. The domain of each \(y_i\), in a given subproblem, is set to either \(\text{left}_i\) or \(\text{right}_i\). Note that, all subproblems are smaller in size than the original one.

5.3.2 Identification of promising subproblems

Depending on the form of the objective function and by reasoning on the feasible solution used to decompose an original problem, one can identify certain subproblems that constitute more promising parts of the search space than others. If one explores those subproblem first, then the chances of finding better feasible solutions are higher. This process may speed up the time to finding the optimal solution.

For instance, consider a minimization COP \((X, D, C, f)\) where the objective function is linear of the form \(\sum_i w_i y_i\) and \(w_i \geq 0\). Suppose we have three variable \(y_1, y_2, y_3 \in \{1, 2, 3, 4\}\) and all \(w_i\) are 1. Consider a feasible solution \(S\) in which \(y_1\) is 2, \(y_2\) is 3, and \(y_3\) is 3. There are 8 subproblems based on this feasible solution:

- (a) \(y_1 \in \{1, 2\}\), \(y_2 \in \{1, 2, 3\}\), and \(y_3 \in \{1, 2, 3\}\)
- (b) \(y_1 \in \{1, 2\}\), \(y_2 \in \{1, 2, 3\}\), and \(y_3 \in \{4\}\)
- (c) \( y_1 \in \{1, 2\}, \ y_2 \in \{4\}, \ \text{and} \ y_3 \in \{1, 2, 3\} \)

- (d) \( y_1 \in \{1, 2\}, \ y_2 \in \{4\}, \ \text{and} \ y_3 \in \{4\} \)

- (e) \( y_1 \in \{3, 4\}, \ y_2 \in \{1, 2, 3\}, \ \text{and} \ y_3 \in \{1, 2, 3\} \)

- (f) \( y_1 \in \{3, 4\}, \ y_2 \in \{1, 2, 3\}, \ \text{and} \ y_3 \in \{4\} \)

- (g) \( y_1 \in \{3, 4\}, \ y_2 \in \{4\}, \ \text{and} \ y_3 \in \{1, 2, 3\} \)

- (h) \( y_1 \in \{3, 4\}, \ y_2 \in \{4\}, \ \text{and} \ y_3 \in \{4\} \)

Since the objective function is linear in this case, one might think of different heuristics that may be adopted in order to compare two subproblems in terms of which one has higher chances of improving on \( f(S) \). For example, for each subproblem we may sum the minimum of each decision variable \( y_i \) and sort the subproblems in decreasing order by this sum. The rationale behind this method is that if we find a solution in the subproblem that has the smallest sum, then that solution would be very tight. We refer to this heuristic as \( \text{minmin} \). Conversely, one might think in terms of the maximum values in each \( y_i \) instead of the minimum values. This gives us another heuristic method that we call \( \text{minmax} \). Finally, a simple method is to count the number of \( y_i \)'s in each subproblems whose domain values are less than or equal to the the value of \( y_i \) in \( S \). i.e., those \( y_i \)'s whose domain is \( left_i \) will not get worse values in the subproblem than the values in \( S \) and hence the more \( y_i \)'s that fit this criterion, the better chances we have in improving the objective function value. We refer to this last method as \( \text{maxleft} \).

We could also develop problem-specific heuristics for objective functions that are linear or of any other form that may yield better orderings tailored to the problem at hand.

5.3.3 Strengthening the cost-based filtering

The solution-based decomposition of the original problem generates subproblems of smaller size. Furthermore, some of these subproblems may be more promising
in terms of finding better feasible solution than others. One can exploit these subproblems in order to strengthen the cost-based filtering in the following way.

First, the value of the solution of the original problem can be used as a bounding mechanism in order to eliminate certain subproblems that does not bring any improvements. In other words, we can avoid visiting any subproblem whose lower bound is worse than the value of the feasible solution $S$ (the current best upper bound). For instance, in our example, $f(S) = 2 + 3 + 3 = 8$ and since the objective function is linear, a lower bound can be simply computed by summing the minimums of the $y_i$'s in each subproblem. Now, it is easy to see that the lower bound of subproblems (d), (f), (g), and (h) are greater than or equal to $f(S)$, the current upper bound. Hence, these subproblems shall not be visited at all.

Second, if we start exploring the most promising subproblems first, the chances of improving on the best upper bound ($f(S)$) are higher. This in turn tightens the best upper bound for the next subproblems which leads to more pruning of the next subproblems. For instance, suppose we solve subproblem (a) and find a better feasible solution where each $y_i = 2$. The value of the best upper bound is now 6 instead of 8. Then we can also avoid visiting subproblems (b) and (c) since their lower bound is 6. Thus, we are only left with subproblem (e) whose lower bound is 5 to explore.

Finally, when we visit a subproblem, we may add a constraint that the value of a feasible solution must be better than the value of the best upper bound found so far. For instance, if the subproblem (e) does not have a feasible solution with value 5, the added constraint will speed up the process of proving the infeasibility of (e) since our best upper bound is 6.

### 5.3.4 Decomposition-based search

The final characteristic of the basic $S&D$ is to recursively decompose the subproblems in a depth-first manner by visiting the most promising subproblems first. The root node of such a tree is the original problem. Each internal node is one of the
subproblems of the parent node. At each internal node, we either find a feasible
solution $S$ or we fail. If we find a feasible solution $S$, we update our best upper
bound ($\text{upperB}$) and use the solution $S$ to sequentially generate the subproblems
in increasing order of promise (using one the heuristics discussed before) and re-
curse. If we fail, then we backtrack to the parent node and recurse on the next
most promising subproblem. There are two ways in which we can fail at an inter-
nal node. Before we try to solve the subproblem, we fail if the current $\text{upperB}$ is
smaller than a valid lower bound of the subproblem. If, however, the lower bound
is smaller than the current $\text{upperB}$, we modify our CSP by adding a constraint
that states that all solutions must have an objective value smaller than $\text{upperB}$.
If the subproblem is infeasible, we also fail. When we backtrack to the root node,
the value $\text{upperB}$ constitutes the value of the optimal solution. Note that if the
decomposition process reach a specified depth, we do not decompose the subprob-
lems anymore but we solve them to optimality using $B&B$ (or any other exact
method).

The pseudo-code for the basic $S&D$ is show in Algorithm 2. It takes as input two
parameters: optimization problem $oPb = (X, D, C, f)$ and the depth limit $\text{depth}$
and outputs the optimal solution ($\text{bestSol}$) along with its objective function value
$\text{upperB}$. At line 3, we set the global variable $\text{upperB}$ to infinity whereas at line 4
we let $\text{pb}$ the CSP corresponding to $oPb$. The final step, in line 5, computes the
$\text{bestSol}$ by calling the function $\text{solveAndDecompose}$ shown in Algorithm 3.

The function $\text{solveAndDecompose}$ has four parameters: the csp $\text{pb}$, the function
$f$, the depth limit $\text{depth}$, and the initial 0. At line 2, if the current best $\text{upperB}$
is smaller than or equal to a valid lower bound of $\text{pb}$ computed by calling function
$\text{lowerBound()}$, we fail. Note that, when the objective function is linear, a valid
lower bound can easily be computed by summing the minimum values of the $y_i$’s.
If we do not fail, then, at line 3, we add to $\text{pb}$ a constraint that enforces that any
feasible solution shall improve on $\text{upperB}$.

Next, at line 4, we test whether we reached our depth limit or not. In case, we
reached the depth limit, at lines 15, we solve the subproblem to optimality and
update the best upper bound if a solution is found (at lines 16-18). Otherwise, at line 5, we solve pb.

If no solution is found, we fail. Otherwise, we update our upperB (at line 7) and our bestSol (at line 8). Then, at line 9, by using the feasible solution found (sol), we decompose pb in a sequential manner (line 10) and recurse on every subproblem subPb of pb at line 11.

Note that, at line 10, the function generateNextPromosingSubproblem must generate the subproblems in the order dictated by one of the heuristics discussed above (i.e., minmin, minmax, and minleft) one by one in order to avoid generating an exponential number of them. Unfortunately, it seems quite challenging to implement such a function for the heuristics minmin and minmax because the ordering depends on the values of the domains. Luckily, for minleft, we can generate the subproblems in the preferred order one at a time. The basic idea is to generate the subproblem where the domains of all the y_i’s are left_i. Then, all but one have domain left_i. Then, all but two have domain left_i, and so on. A simple tail-recursive procedure can do achieve this. For instance, if we want all subproblems in which k out of the n y_i variables have domain left_i and the rest domain right_i, we can do the following:

1. set all variables’ domains to left_i if k = n (base case 1);
2. set all variables’ domains to right_i if k = 0 (base case 2);
3. set the domain of the first variable i to left_i and recurse on the remaining n − 1 variables with k;
4. set the domain of the first variable i to right_i and recurse on the remaining n − 1 variables with k − 1;
5.3.5 Example

Suppose we have the following COP: \( \min \sum_i y_i \) \( \text{s.t.} \ y_1 + y_2 \geq 3, y_1 + y_3 \geq 3, y_2 + y_3 \geq 3 \) where \( y_i \in \{1, \ldots, 5\} \)

Figure 5.1 shows a trace of the full search tree using the basic S&D to find the optimal solution (\( \text{bestSol} = (y_1 = 1, y_2 = 2, y_3 = 2) \)). The search tree generated by S&D is of depth 2. The green nodes present a feasible solution obtained by the function \( \text{solve} \). Since, we have three variables in the objective function, at each depth we have \( 2^3 = 8 \) subproblems to explore. At the root node, we find a feasible solution with objective value 8 which we use to decompose the problem. At depth 1, we explore the most promising subproblem \( \text{subPbl1} \) according to our heuristic \( \text{maxleft} \). We find a better feasible solution that improves the objective function value to 5 (which is the optimal solution). We decompose \( \text{subPbl1} \) using the feasible solution found and start exploring at depth 2 \( \text{subPbl11} \). However, \( \text{subPbl11} \) is infeasible because it cannot improve on the value of \( \text{upperB} \). We backtrack and explore the next most promising subproblem \( \text{subPbl12} \). But, this time, we immediately fail because the lower bound of \( \text{subPbl12} \) is 5 which means we cannot improve \( \text{upperB} \). So, we backtrack and explore the next subproblems. For the remaining subproblems at depth 2, we either fail immediately (denoted by squared X) or the problem is shown infeasible (denoted by circled X). After we explore all subproblem at depth 2, we backtrack and explore the remaining subproblems at depth 1. For all these subproblems, we immediately fail.

This simple example clearly shows the essence of the solution-based decomposition search to prune some sub-optimal parts of search tree and the efficiency in which we find the optimal solution.
5.4 Improving Solve and Decompose

The basic \( S&D \) can has many benefits in terms of pruning large parts of search tree, generating subproblems of smaller size, identifying the promising subproblems and consequently finding fast the optimal solution. But it has a main drawback presented by the difficulty of proof of optimality due to the number of generated subproblems which is exponential in the number of variables appearing in the objective function. This means that the generated search tree has a large branching factor.

To remedy this problem, we propose limiting the number of visited subproblem to a certain level at which we hope that the optimal solution (or a good solution) is found. Next, instead to traverse all the rest subproblems, we propose to return on the parent node and to solve it to optimality.

Also, it should be noted that the subproblems are ranked in levels according to the number of \( left_i \) subdomains. We noted each level by \( L_p \) where \( p \) presented the number of \( left_i \) subdomains. Then, we propose limiting the decomposition process to a level \( L_p \) such that all subproblems belonging to levels \( L_q \), \( 0 \leq q < p \) are infeasible. That is mean that the decomposition process is stopped after finding a solution at level \( L_p \) and not in previous levels.

The proposed modifications to the \( \text{solveAndDecompose} \) constitute our new \( S&D \) algorithm that we refer to as improved \( S&D \). The difference to the basic \( S&D \) is that we now limit the number of visited subproblems, line 12 of the Algorithm 4. We use a boolean variable \( stop \) to test whether the \( upperB \) is improved or not at a level \( L_p \). In case, the \( upperB \) is improved, the variable \( stop \) is used to stop the while loop. That is mean we stop the decomposition process, and at line 23, we solve the parent problem \( pb \) (the original problem) to optimality. Otherwise, if any solution is not found, we increment \( p \) line 17 to move to the next level \( p+1 \). This process is repeated until the \( upperB \) is improved or \( p \) reaches \( |Y| \) (ie. \( pb \) is not feasible). Note that, at line 14, the function \( generateNextPromosingSubproblem \) must generate subproblems belonging to the level \( L_p \), where \( p \) is an input parameter.
of the function, one by one in order to avoid generating an exponential number of them.

5.5 Computational results

5.5.1 Benchmark problems

For our computational study, we consider two COPs: the multidimensional multiple-choice knapsack problem (MMKP) Moser et al. (1997b) and the Steel Mill Slab Design Problem (SMSDP) Frisch et al. (2001).

5.5.1.1 MMKP

MKKP is a generalization of the classical knapsack problem. MMKP is defined by a set of items divided into \( n \) groups and a resource with \( m \) dimensions. The resource has a limited capacity in each dimension \( k \) denoted by \( c_k \). Each item \( j \) belonging to a group \( G_i \) has profits \( p_{i,j} \) and a weight in each dimension, denoted by \( w_{k,i,j} \). The objective is to select exactly one item from each group such that their overall profit is maximized, while the overall weight does not exceed any resource capacities. A model for the MMKP is shown in Figure 5.2.

Decision variable \( item_i \) is introduced for each group \( G_i \) whose value corresponds to the index of selected item of this group. \( profit_i \) corresponds to the profit of the selected item of each group whereas \( weight_{k,i} \) is the weight of the selected item of each group for each dimension \( k \). The first set of constraint enforce that the capacity in each dimension is not exceeded. The second (third) set of constraints compute the profit of the selected item (the weight of the selected item in each dimension).
5.5.1.2 SMSDP

The SMSDP Frisch et al. (2001) consists in assigning \( n \) orders to a set of slabs. Each order has a color and a weight representing the slab capacity it takes. Each slab has a capacity that must be chosen from the increasing set of capacities \( \{c_1 \ldots c_m\} \) where \( m \) is the number of slab capacities. A solution is an assignment of orders to slabs such that the total weights of the orders in a slab must not exceed the slab capacity and at most \( p \), of \( k \) total colors, different colors are present in each slab. The objective is to minimize the sum of the weights of the slabs used in the solution.

We use the model in Frisch et al. (2001) and shown in Figure 5.3. We introduce for each slab \( j \), a variable \( \text{slab}_j \) whose value corresponds to the capacity that takes. 0 is added to the domain of each slab variable in case when the slab is unused. We introduce \( 1,0,0 \) two binary variables \( \text{order}_{i,j} \) and \( \text{color}_{i,j} \). \( \text{order}_{i,j} \) is equal to 1 iff order \( i \) is assigned to slab \( j \) whereas \( \text{color}_{i,j} \) is equal to 1 if color \( i \) is assigned to slab \( j \), 0 otherwise. The first set of constraints guarantees that slab capacities are not exceeded. The second set of constraints enforce that the number of distinct colors per slab shall not exceed \( p \). The third set of constraints channel between the color matrix and the order matrix. Whenever an order is assigned to a slab, then the color of that order is added to the colors of that slab. Constraints (4) break the symmetry among the slab variables whereas constraint (5) adds a valid lower bound.

5.5.2 Settings

For the MMKP, we propose a set of random instances with varying sizes, using the proposed method by Han et al. (2010b), where profits (respectively weights) are generated using the uniform generating function (respectively the uncorrelated generating function). Each instance can be described by three parameters, the number of groups \( n \), the number of items for each group \( |G_i| \) and the number of resource dimensions \( m \). To show the behavior of S&D when varying the number
of domain values, we classify the proposed instances in two classes. The first class (respectively the second class) is described by a fixed number of items ($|G_i| = 20$) (respectively $|G_i| = 50$). For each class, we use a growing number of groups $n = 10, 15, 20, 25$ to show the impact of S&D when varying the number of objective variables which used in the decomposition process. For both classes, the number of dimensions is fixed to 10 and the weight values are selected randomly between 0 and $R = 100$. Ten instances were generated randomly for each combination of parameters, for a total 80 instances.

For the SMSDP, we generate randomly a set of instances as subsets of the real instance available at csplib (prob038) with 10, 11, 12, 13, 14 and 15 orders, since the real instance with 111 orders is a quite large instance and it is hard to solve with the model in Frisch et al. (2001). Note that, slab capacities and colors are the same as in the CSPLib real instance. Five instances were generated for each order, for a total 30 instances.

Our experiments are carried out on a machine with Pentium(R) Dual-Core CPU 2.6 GHz processor, with 2 GB RAM. As our constraint programming solver we use the Choco Solver, version 2.1.1 \footnote{http://www.emn.fr/z-info/choco-solver/} for solving CSPs as well as COPs. Variable and value selection heuristics were selected as it were defined by the default branching heuristic of Choco. We set the time limit for each instance to 3600 seconds.

### 5.5.3 Results

The objectives of our experiments are aimed at answering the following questions:

- Does the proposed order of the generated subproblems is able to identify the promising ones?
- What is the behavior of the basic S&D algorithm in both terms search and proof of optimality?
Chapter 4. Solving Constrained Optimization Problems By Solution-based Decomposition Search

• Does the improved $S&D$ algorithm cancel the drawbacks of the basic $S&D$ algorithm?

• Does the improved $S&D$ algorithm bring any benefit in terms of pruning and of search efficiency compared to the $B&B$ algorithm?

• What effect can we observe when we vary the depth limit of the decomposition?

The results of basic $S&D$, improved $S&D$ and $B&B$ algorithms of the first and second class of MMKP instances (respectively the SMSDP instances) are presented in Table 5.1 and 5.2 (respectively Table 5.3). In order to show the ability of $S&D$ to identify the promising subproblems, we start the evaluation by setting the depth parameter $depth$ to 1.

For the basic $S&D$, we report in column $L_p$ the location level $L_p$ of the optimal solution for each instance. We observe that the optimal solution is found in a reduced location level regardless the number of objective variables. The algorithm has almost the same behavior when we vary the number of objective variable. In fact, for the MMKP, the position median location level of the optimal solution in Table 5.1 (respectively Table 5.2) is equal to 0, 1, 0.5 and 2 (respectively 0, 0.5, 0.5 and 1) for respectively 10, 15, 20 and 25 objective variables. Also, we observe that the basic $S&D$ has a similar behavior regardless the domain size of objective variables (Table 5.1 and Table 5.2).

For the SMSDP, the position median location level of the optimal solution is equal to 0, 1, 0, 0, 0, 0 for respectively 10, 11, 12, 13, 14 and 15 objective variables. Based on the results, we clearly observe the performance of $S&D$ to identifying and exploring the promising subproblems.

In order to further understand the behavior of the basic $S&D$, we report in column $CPU$ the time (in seconds) required by $S&D$, the time to find the optimal solution column $FindObj$. For both MMKP and SMSDP, results show the efficiency of the basic $S&D$ to find the optimal solution. On the other side results show a main
Chapter 4. Solving Constrained Optimization Problems By Solution-based Decomposition Search

weakness is that we spend most of time trying to prove optimality. For example, the basic $S&D$ is not able (reach the time limit) to prove the optimality when the number of objectives variables is greater or equal to 20. Note that all instances show a very similar behavior in terms of finding the optimal solution and prove optimality.

In the figure 5.4, we report the the CPU time consumed by the basic $S&D$ in each level $L_p, p = 0, \ldots, |Y| = 15$ for a representative instance $inst17$ of MMKP. Note that all other instances show a very similar behavior as instance $inst17$.

Although, the optimal solution is found at level $L_1$, we remark that the basic $S&D$ spent a lot of time due to the exponential number of the generated subproblems. Also, we remark that it spent a lot of time at level $L_{|Y|/2}$ since it contains the biggest number of subproblems.

As we conclude from the previous that the basic $S&D$ to be efficient, it has to be effective in term of proof of optimality.

Now, we analyse the improved $S&D$. We show if it is able to remedy the spend of time trying to prove optimality. We report in column $L_p$ and $valSol$ the position and the value of the founded solution at level $L_p$. The quality of a solution is measured by the percentage gap between the optimal solution value and the founded solution value at level $L_p$, i.e.$100 \times (optimal\_solution\_value - found\_solution\_value)/(optimal\_solution\_value)$. We observe that the optimal solution is found in $81.25\%$ for the MMKP instances (respectively $76.7\%$ for the SMSDP instances) and the percentage gap between the found and the optimal solution is very small and it is equal to $0.09$ (respectively $0.24$) in average.

Also, we observe that the proposed parameter plays a significant role to reduce the CPU time. In fact, results show that the time spent by $S&D$ after finding the solution at level $L_p$ is clearly reduced compared with the basic $S&D$. In this respect, the proposed modification plays a central role to best fine tune $S&D$.

Finally, we show the results obtained by $B&B$. Column $CPU$, $Backtracks$ and $Nodes$ show the runtime respectively the number of visited nodes and the number
of backtracks of \( B&B \). The results clearly show that \( S&D \) improves on \( B&B \) in terms of pruning and of search efficiency in all instances. These improvements may reach 3 orders of magnitude.

In the experiments presented above, by setting the depth limit to 1, we showed that the improved \( S&D \) outperform \( B&B \) algorithm in terms of pruning and of search efficiency. We now investigate if the depth parameter plays any role to improve the founded results.

For a depth value greater than 1, experiments (see Table 4 and Table 5) showed that the depth parameter plays a negative role to improve the founded results in general, and when we increase the depth value the performance goes down. Since the size of the subproblems decrease with larger depth, we may end up with less promising subproblems to explore first and we may still be decomposing a non-promising subproblem. Also, when we increase depth, the size of the subproblems decrease and consequently the number of feasible solution decrease and in many cases the subproblem contains a single solution. Therefore, in this case we must traverse all generated subproblems such that any of them is inconsistent. This explain the behavior of the depth parameter in many instances.

In some instances such as instances ins5, ins7, and ins30 of SMSDP and inst39 inst41 and inst59 MMKP, we remark that the performance may go up and down and is rather unstable. We observe a non-consistent behavior when we increase depth. Sometimes we get lucky and explore a promising subproblem first and the performance is improved with smaller depth.

Based on the results, we can identify that the configuration of \( S&D \) when depth equal to 1 can be viewed as the best configuration.

### 5.6 Related work

In this section, we discuss the relevant works that have been done on decomposition search that are closely related to our works.
The closest work to ours is the one of Milano and van Hoeve (2004) which generalizes Milano and Hoeve (2002). Hoeve and Milano (2004) introduces a search strategy called Decomposition Based Search (DBS) where a domain splitting strategy is employed to break down a finite domain problem into subproblems. DBS is used for solving both CSPs and COPs and it is based on two steps: subproblem generation and subproblem solution. The generation of subproblems is done through domain splitting. Domain splitting is used to decompose the original problem into a set of subproblems with smaller sub-domains according to a value ordering heuristic. The heuristic needs to rank a value with two levels of accuracy: first, it should measure accurately how successful a value is; second, it is required to discriminate among values with the same rank.

They employ Limited Discrepancy Search (Harvey and Ginsberg, 1995) to exploit the ranks. At a specified level $d$, defined by a user in a pre-search stage, the domains are not split anymore and the resulting subproblem is solved. The authors propose two alternatives to solve the subproblem. The first one consists of using the Depth-First Search for solving subproblems at level $d$ and the second consists of applying again the principle of DBS but by using another value ordering heuristic at each recursive call. But, only the evaluation of the first strategy is carried out in Hoeve and Milano (2004) since implementing the DBS recursively is quite challenging. It is useless to use the same heuristic for the subproblem solution since all values belonging to the subproblem have a very similar rank. Thus, it is advised to use another value ordering heuristic which shall partition further the sub-domains which might be difficult to have for the same problem for each recursive call.

DBS is close to $S&D$. In fact, Algorithm 3.1 of DBS (presented in Hoeve and Milano (2004)) can be viewed as the $S&D$ algorithm where the depth parameter $d$ of DBS represents the number of objective variables of $S&D$ and the parameter $depth$ of $S&D$ is set to 1. But, unlike, DBS, which uses value ordering heuristics to generate the subproblems, $S&D$ uses a feasible problem to the CSP of the COP. It is this feature which makes $S&D$ very simple when it comes to implementation as opposed to DBS. Finally, $S&D$ uses only a subset of the variables appearing in the
objective function in the decomposition that will be used to identify the promising subproblems. However, in DBS, all variables can be used in the decomposition.

The work in Kitching and Bacchus (2009) introduced a limited approach of exploiting decomposition on certain types of COPs in which the constraints and objective function are decomposable. Their approach is compatible with a fully flexible branch and bound search employing dynamic variable and value ordering. In practical terms, their method requires that the variables appearing in the objective function be decomposed into a set of sub-objective variables so that each subset and each constraint only depends on a proper subset of the variables.

The work in Régin et al. (2014) described an efficient parallel version of the decomposition for solving CSPs. They propose an Embarrassingly Parallel Search (EPS) method. EPS decomposes the original problem in large distinct subproblems, each one is then solved independently by workers. This method uses the cooperation between computation units (workers) to divide the work dynamically during the resolution. It splits statically the initial problem into a large number of subproblems that are consistent with the propagation and puts them in a queue. Once this decomposition is over, the workers take dynamically the subproblems from the queue. The solving process ends when all subproblems are solved.

### 5.7 Conclusion

In this chapter, we propose S&D which a systematic iterative depth-first strategy that is based on problem decomposition. S&D uses a feasible solution at each node in order to decompose the problem into smaller subproblems. The same solution is also used in order to explore subproblems that have more promise in finding better solutions as well as a bound for the next subproblems. The number of subproblems is bounded and is controlled by parameter $p$ whereas their size is controlled by depth which is a depth limit after which we stop the decomposition process. S&D is designed so (i) to speed up the time to finding the optimal solution by problem decomposition and visiting of promising subproblems first; and (ii) to speed up
the proof of optimality by strengthening the cost-based filtering. Our experiments on MMKP and SMSDP benchmarks show that S&D improves on B&B in general and the improvement may reach orders of magnitude.
Chapter 4. Solving Constrained Optimization Problems By Solution-based Decomposition Search
Chapter 4. Solving Constrained Optimization Problems By Solution-based Decomposition Search

Algorithm 2: Basic S&D

\[\text{input : } \alpha \text{Pb} = (X, D, C, f); \text{ depth : Integer; }\]
\[\text{output: } \text{bestSol : Solution; } \text{upperB : Integer; }\]
\[\text{begin}\]
\[\quad \text{upperB is a global variable; }\]
\[\quad \text{upperB } \leftarrow \infty;\]
\[\quad \text{pb } = (C, D, X);\]
\[\quad \text{bestSol } \leftarrow \text{solveAndDecompose}(\text{pb}, f, \text{depth}, 0);\]
\[\text{end}\]

Algorithm 3: solveAndDecompose(pb, f, depth, levelID)

\[\text{input : } \text{pb } = (X, D, C) : \text{CSP; } f : \text{function; }\]
\[\text{output: } \text{bestSol : Solution; }\]
\[\text{begin}\]
\[\quad \text{if } \text{upperB } > \text{lowerBound}(\text{pb}) \text{ then}\]
\[\quad \quad \text{add } f(Y) < \text{upperB to constraints of pb; }\]
\[\quad \quad \text{if } \text{levelD } \leq \text{depth then}\]
\[\quad \quad \quad \text{sol } \leftarrow \text{solve(pb); }\]
\[\quad \quad \quad \text{if } \text{sol } \neq \text{null then}\]
\[\quad \quad \quad \quad \text{upperB } \leftarrow f(\text{sol});\]
\[\quad \quad \quad \quad \text{bestSol } \leftarrow \text{sol; }\]
\[\quad \quad \quad \text{while } \text{pb has more subproblems do}\]
\[\quad \quad \quad \quad \text{subPb } \leftarrow \text{generateNextPromisingSubproblem( pb, sol); }\]
\[\quad \quad \quad \quad \text{solveAndDecompose( subPb, f, levelID } + 1);\]
\[\quad \quad \text{end}\]
\[\quad \text{end}\]
\[\quad \text{else}\]
\[\quad \quad \text{sol } \leftarrow \text{optimize(pb, f); }\]
\[\quad \quad \text{if } \text{sol } \neq \text{null then}\]
\[\quad \quad \quad \text{upperB } \leftarrow f(\text{sol});\]
\[\quad \quad \quad \text{bestSol } \leftarrow \text{sol; }\]
\[\quad \text{end}\]
\[\text{end}\]
\[\text{return } \text{bestSol;}\]

Algorithm 4: improvedSolveAndDecompose(pb, f, depth, levelID)
Chapter 4. Solving Constrained Optimization Problems By Solution-based Decomposition Search

Figure 5.1: A Trace of the basic S&D on an example.

Constraints:
\( \sum_{i=1}^{n} \text{weight}_{k,i} \leq c_k \ \forall k \in \{1, \ldots, m\} \)
\( \text{profit}_i = p_{i,\text{item}} \ \forall i \in \{1, \ldots, n\} \)
\( \text{weight}_{k,i} = w_{k,i,\text{item}} \ \forall k \in \{1, \ldots, m\}, \forall i \in \{1, \ldots, n\} \)

Decision variables and domains:
\( \text{profit}_i \in \{p_{ij} | \forall j \in (1, \ldots, |G_i|)\}, \ \forall i \in \{1, \ldots, n\} \)
\( \text{item}_i \in \{1, \ldots, |G_i|\}, \ \forall i \in \{1, \ldots, n\} \)
\( \text{weight}_{k,i} = \{w_{k,i,j} | \forall j \in (1, \ldots, |G_i|)\}, \ \forall i \in \{1, \ldots, n\}, \forall k \in \{1, \ldots, m\} \)

Objective function:
maximize \( \sum_{i=1}^{n} \text{profit}_i \)

Figure 5.2: Multidimensional multiple choice knapsack problem formulation.

Constraints:
\( \sum_{i=1}^{n} \text{weight}(i) \times \text{order}_{i,j} \leq \text{slab}_j \ \forall j \in \{1, \ldots, n\} \)
\( \sum_{i=1}^{k} \text{color}_{i,j} \leq p_i \ \forall j \in \{1, \ldots, n\} \)
\( (\text{order}_{i,j} = 1) \rightarrow (\text{color}_{\text{color}(i,j)} = 1), \ \forall i, j \in \{1, \ldots, n\} \)
\( \text{slab}_j \leq \text{slab}_{j+1}, \ \forall j \in \{1, \ldots, n-1\} \)
\( \sum_{i=1}^{n} \text{weight}(i) \leq \sum_{j=1}^{n} \text{slab}_j \)

Decision variables and domains:
\( \text{slab}_j \in \{0, c_1, \ldots, c_m\}, \ \forall j \in \{1, \ldots, n\} \)
\( \text{order}_{i,j} \in \{0, 1\}, \ \forall i, j \in \{1, \ldots, n\} \)
\( \text{color}_{i,j} \in \{0, 1\}, \ \forall i \in \{1, \ldots, k\}, \forall j \in \{1, \ldots, k\} \)

Objective function:
minimize \( \sum_{j=1}^{n} \text{slab}_j \)

Figure 5.3: Steel Mill Slab Design Problem formulation.
Figure 5.4: Time consumed by $S&D$ in each level for MMKP instance inst17
### Table 5.1: The runtime in seconds for MMKP instances using basic SKD and improved SKD as well as means using B&B. Note that all runs have been reached.

<table>
<thead>
<tr>
<th>Instance</th>
<th>Basic SKD</th>
<th>Improved SKD</th>
<th>B&amp;B</th>
<th>OAP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inst1</td>
<td>10</td>
<td>70.37</td>
<td>7.78</td>
<td>7.78</td>
</tr>
<tr>
<td>Inst2</td>
<td>0</td>
<td>86.96</td>
<td>78.57</td>
<td>78.57</td>
</tr>
<tr>
<td>Inst3</td>
<td>0</td>
<td>14.58</td>
<td>84.35</td>
<td>84.35</td>
</tr>
<tr>
<td>Inst4</td>
<td>0</td>
<td>86.36</td>
<td>85.00</td>
<td>85.00</td>
</tr>
<tr>
<td>Inst5</td>
<td>0</td>
<td>79.17</td>
<td>86.11</td>
<td>86.11</td>
</tr>
<tr>
<td>Inst6</td>
<td>0</td>
<td>90.91</td>
<td>87.50</td>
<td>87.50</td>
</tr>
<tr>
<td>Inst7</td>
<td>0</td>
<td>70.83</td>
<td>87.50</td>
<td>87.50</td>
</tr>
<tr>
<td>Inst8</td>
<td>0</td>
<td>68.00</td>
<td>88.24</td>
<td>88.24</td>
</tr>
<tr>
<td>Inst9</td>
<td>0</td>
<td>90.91</td>
<td>92.59</td>
<td>92.59</td>
</tr>
<tr>
<td>Inst10</td>
<td>0</td>
<td>100.00</td>
<td>100.00</td>
<td>100.00</td>
</tr>
<tr>
<td>Inst11</td>
<td>15</td>
<td>83.50</td>
<td>92.84</td>
<td>92.84</td>
</tr>
<tr>
<td>Inst12</td>
<td>1</td>
<td>95.56</td>
<td>95.31</td>
<td>95.31</td>
</tr>
<tr>
<td>Inst13</td>
<td>1</td>
<td>93.99</td>
<td>97.67</td>
<td>97.67</td>
</tr>
<tr>
<td>Inst14</td>
<td>1</td>
<td>99.84</td>
<td>98.77</td>
<td>98.77</td>
</tr>
<tr>
<td>Inst15</td>
<td>1</td>
<td>97.53</td>
<td>96.47</td>
<td>96.47</td>
</tr>
<tr>
<td>Inst16</td>
<td>1</td>
<td>98.98</td>
<td>97.49</td>
<td>97.49</td>
</tr>
<tr>
<td>Inst17</td>
<td>1</td>
<td>99.12</td>
<td>97.05</td>
<td>97.05</td>
</tr>
<tr>
<td>Inst18</td>
<td>1</td>
<td>98.26</td>
<td>96.97</td>
<td>96.97</td>
</tr>
<tr>
<td>Inst19</td>
<td>1</td>
<td>99.97</td>
<td>99.19</td>
<td>99.19</td>
</tr>
<tr>
<td>Inst20</td>
<td>1</td>
<td>99.99</td>
<td>99.31</td>
<td>99.31</td>
</tr>
<tr>
<td>Inst21</td>
<td>20</td>
<td>99.99</td>
<td>99.54</td>
<td>99.54</td>
</tr>
<tr>
<td>Inst22</td>
<td>2</td>
<td>97.81</td>
<td>95.18</td>
<td>95.18</td>
</tr>
<tr>
<td>Inst23</td>
<td>1</td>
<td>99.47</td>
<td>94.14</td>
<td>94.14</td>
</tr>
<tr>
<td>Inst24</td>
<td>2</td>
<td>99.12</td>
<td>94.14</td>
<td>94.14</td>
</tr>
<tr>
<td>Inst25</td>
<td>2</td>
<td>97.53</td>
<td>96.47</td>
<td>96.47</td>
</tr>
<tr>
<td>Inst26</td>
<td>2</td>
<td>99.25</td>
<td>98.25</td>
<td>98.25</td>
</tr>
<tr>
<td>Inst27</td>
<td>2</td>
<td>98.61</td>
<td>98.61</td>
<td>98.61</td>
</tr>
<tr>
<td>Inst28</td>
<td>2</td>
<td>99.97</td>
<td>99.97</td>
<td>99.97</td>
</tr>
</tbody>
</table>
Table 5.2: The runtime in second for MMKP instances using basic S&D and improved S&D as well as using B&B. Note that "-" means that the time limit has been reached.
## Chapter 4. Solving Constrained Optimization Problems by Solution-based Decomposition Search

### Improved \& Basic \& Decomposer

| \( L_p \) | \( \text{Obj} \) | \( \text{CPU} \) | \( \text{FindObj} \) | \( \text{Backtracks} \) | \( \text{Nodes} \) | \( \text{Obj} \) | \( \text{CPU} \) | \( \text{FindObj} \) | \( \text{Backtracks} \) | \( \text{Nodes} \) | \( \text{Obj} \) | \( \text{CPU} \) | \( \text{FindObj} \) | \( \text{Backtracks} \) | \( \text{Nodes} \) |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 10 | 209 | 19 | 10 | 30560 | 113137 | 0 | 209 | 209 | 10 | 10 | 30560 | 113137 | 209 | 572 | 291885 | 975028 | 81.48 | 98.25 |
| 0 | 209 | 198 | 11 | 30469 | 112114 | 0 | 209 | 209 | 116 | 116 | 30469 | 112114 | 209 | 498 | 3213152 | 890107 | 26.58 | 76.71 |
| 1 | 187 | 53 | 11 | 60207 | 116654 | 0 | 197 | 197 | 37 | 8 | 204625 | 448493 | 187 | 251 | 1567333 | 3906182 | 30.19 | 85.26 |
| 0 | 201 | 63 | 19 | 98809 | 199586 | 0 | 201 | 201 | 19 | 19 | 98809 | 199586 | 201 | 102 | 642654 | 1610182 | 69.84 | 81.37 |

### Table 5.3: The runtime in seconds for SAMBP instances using basic \& improved \( \text{SD} \) and \( \text{D} \) as well as using \( \text{B} \& \text{B} \). Note that "... means that the time limit has been reached.
### Table 5.4: The runtime of $S&D$ in second for MMKP instances using different depth. Note that “–” means that the time has reached 1000s.

<table>
<thead>
<tr>
<th>depth</th>
<th>First class</th>
<th>Second class</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>ins1</td>
<td>8</td>
<td>10</td>
</tr>
<tr>
<td>ins2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>ins3</td>
<td>41</td>
<td>45</td>
</tr>
<tr>
<td>ins4</td>
<td>3</td>
<td>12</td>
</tr>
<tr>
<td>ins5</td>
<td>5</td>
<td>7</td>
</tr>
<tr>
<td>ins6</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>ins7</td>
<td>7</td>
<td>12</td>
</tr>
<tr>
<td>ins8</td>
<td>8</td>
<td>7</td>
</tr>
<tr>
<td>ins9</td>
<td>2</td>
<td>15</td>
</tr>
<tr>
<td>ins10</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>ins11</td>
<td>132</td>
<td>216</td>
</tr>
<tr>
<td>ins12</td>
<td>3</td>
<td>11</td>
</tr>
<tr>
<td>ins13</td>
<td>22</td>
<td>266</td>
</tr>
<tr>
<td>ins14</td>
<td>3</td>
<td>54</td>
</tr>
<tr>
<td>ins15</td>
<td>7</td>
<td>2</td>
</tr>
<tr>
<td>ins16</td>
<td>8</td>
<td>7</td>
</tr>
<tr>
<td>ins17</td>
<td>6</td>
<td>258</td>
</tr>
<tr>
<td>ins18</td>
<td>12</td>
<td>8</td>
</tr>
<tr>
<td>ins19</td>
<td>20</td>
<td>136</td>
</tr>
<tr>
<td>ins20</td>
<td>2</td>
<td>5</td>
</tr>
<tr>
<td>ins21</td>
<td>55</td>
<td>157</td>
</tr>
<tr>
<td>ins22</td>
<td>78</td>
<td>43</td>
</tr>
<tr>
<td>ins23</td>
<td>19</td>
<td>8</td>
</tr>
<tr>
<td>ins24</td>
<td>89</td>
<td>595</td>
</tr>
<tr>
<td>ins25</td>
<td>63</td>
<td>116</td>
</tr>
<tr>
<td>ins26</td>
<td>61</td>
<td>78</td>
</tr>
<tr>
<td>ins27</td>
<td>50</td>
<td>–</td>
</tr>
<tr>
<td>ins28</td>
<td>9</td>
<td>170</td>
</tr>
<tr>
<td>ins29</td>
<td>3</td>
<td>161</td>
</tr>
<tr>
<td>ins30</td>
<td>5</td>
<td>–</td>
</tr>
<tr>
<td>ins31</td>
<td>55</td>
<td>–</td>
</tr>
<tr>
<td>ins32</td>
<td>110</td>
<td>92</td>
</tr>
<tr>
<td>ins33</td>
<td>30</td>
<td>–</td>
</tr>
<tr>
<td>ins34</td>
<td>133</td>
<td>–</td>
</tr>
<tr>
<td>ins35</td>
<td>14</td>
<td>442</td>
</tr>
<tr>
<td>ins36</td>
<td>87</td>
<td>–</td>
</tr>
<tr>
<td>ins37</td>
<td>7</td>
<td>92</td>
</tr>
<tr>
<td>ins38</td>
<td>107</td>
<td>–</td>
</tr>
<tr>
<td>ins39</td>
<td>52</td>
<td>5</td>
</tr>
<tr>
<td>ins40</td>
<td>7</td>
<td>–</td>
</tr>
</tbody>
</table>
Table 5.5: The runtime of S&D in second for SMSDP instances using different depth. Note that “-” means that the time has reached 1000s.

<table>
<thead>
<tr>
<th>depth</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>ins1</td>
<td>10</td>
<td>8</td>
<td>15</td>
<td>67</td>
<td>68</td>
</tr>
<tr>
<td>ins2</td>
<td>116</td>
<td>89</td>
<td>96</td>
<td>102</td>
<td>152</td>
</tr>
<tr>
<td>ins3</td>
<td>37</td>
<td>42</td>
<td>42</td>
<td>40</td>
<td>92</td>
</tr>
<tr>
<td>ins4</td>
<td>19</td>
<td>15</td>
<td>26</td>
<td>31</td>
<td>37</td>
</tr>
<tr>
<td>ins5</td>
<td>120</td>
<td>204</td>
<td>110</td>
<td>141</td>
<td>271</td>
</tr>
<tr>
<td>ins6</td>
<td>70</td>
<td>66</td>
<td>686</td>
<td>644</td>
<td></td>
</tr>
<tr>
<td>ins7</td>
<td>86</td>
<td>92</td>
<td>111</td>
<td>105</td>
<td>227</td>
</tr>
<tr>
<td>ins8</td>
<td>50</td>
<td>69</td>
<td>82</td>
<td>87</td>
<td>193</td>
</tr>
<tr>
<td>ins9</td>
<td>8</td>
<td>4</td>
<td>2</td>
<td>9</td>
<td>14</td>
</tr>
<tr>
<td>ins10</td>
<td>102</td>
<td>77</td>
<td>184</td>
<td>187</td>
<td>187</td>
</tr>
<tr>
<td>ins11</td>
<td>76</td>
<td>301</td>
<td>308</td>
<td>308</td>
<td></td>
</tr>
<tr>
<td>ins12</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>ins13</td>
<td>56</td>
<td>142</td>
<td>383</td>
<td>383</td>
<td>383</td>
</tr>
<tr>
<td>ins14</td>
<td>67</td>
<td>95</td>
<td>142</td>
<td>132</td>
<td>109</td>
</tr>
<tr>
<td>ins15</td>
<td>39</td>
<td>64</td>
<td>134</td>
<td>150</td>
<td>163</td>
</tr>
<tr>
<td>ins16</td>
<td>141</td>
<td>226</td>
<td>117</td>
<td>173</td>
<td>151</td>
</tr>
<tr>
<td>ins17</td>
<td>86</td>
<td>83</td>
<td>123</td>
<td>97</td>
<td>116</td>
</tr>
<tr>
<td>ins18</td>
<td>15</td>
<td>82</td>
<td>27</td>
<td>528</td>
<td>511</td>
</tr>
<tr>
<td>ins19</td>
<td>33</td>
<td>30</td>
<td>26</td>
<td>41</td>
<td>56</td>
</tr>
<tr>
<td>ins20</td>
<td>89</td>
<td>72</td>
<td>71</td>
<td>574</td>
<td>574</td>
</tr>
<tr>
<td>ins21</td>
<td>16</td>
<td>19</td>
<td>21</td>
<td>55</td>
<td>321</td>
</tr>
<tr>
<td>ins22</td>
<td>142</td>
<td>112</td>
<td>193</td>
<td>107</td>
<td>154</td>
</tr>
<tr>
<td>ins23</td>
<td>159</td>
<td>196</td>
<td>188</td>
<td>224</td>
<td>294</td>
</tr>
<tr>
<td>ins24</td>
<td>44</td>
<td>58</td>
<td>419</td>
<td>223</td>
<td>-</td>
</tr>
<tr>
<td>ins25</td>
<td>194</td>
<td>236</td>
<td>334</td>
<td>416</td>
<td>-</td>
</tr>
<tr>
<td>ins26</td>
<td>91</td>
<td>147</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>ins27</td>
<td>92</td>
<td>272</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>ins28</td>
<td>73</td>
<td>431</td>
<td>288</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>ins29</td>
<td>134</td>
<td>146</td>
<td>372</td>
<td>861</td>
<td>601</td>
</tr>
<tr>
<td>ins30</td>
<td>71</td>
<td>41</td>
<td>49</td>
<td>102</td>
<td>73</td>
</tr>
</tbody>
</table>
Chapter 6

Conclusions and Future Work

This chapter summarises the contributions of this thesis and outlines opportunities for future work which we hope will usefully expand the results of this thesis.

6.1 Summary and conclusions

The aim of this thesis is multiple (i) study relationship between a set of knapsack problems (ii) provide a filtering algorithm for the new global constraint \textit{mcmdk} and, design a new strategy for solving constrained optimization problems. This thesis has mainly achieved that goal.

Chapter 3 presents a relationship study between a set of knapsack problems involving dimensions, demands and multiple choice constraints such as: the MKP, the MDMKP, the MCKP, the MMKP and the GUBMKP. This study lead to define the generalized problem called the multiple demand multidimensional multiple choice knapsack problem (MDMMKP) as a generalization of these problems.

And by applying a set of defined transformations between the different integer linear programs of these knapsack extensions, algorithm for that generalization is assumed as a a solver. Evenly, results show that the transformations is able to generate reasonable computing time compared with the original ones.
Chapter 4 presents a new global constraint belonging to the weighted constraints and closely related to the knapsack constraint denoted $mcmdk$. We modeled the $mcmdk$ constraint using the conjunction of sum and implies constraints and we associate it a filtering algorithm "based reasoning”. Experiments show that propagating the $mcmdk$ constraint via the proposed filtering algorithm is more effective and efficient than propagating it using the straightforward conjunction.

Chapter 5 present a new strategy for solving Constrained Optimization Problems (COPs) called solve and decompose (or S&D for short). The proposed strategy is a systematic iterative depth-first strategy that is based on problem decomposition. S&D uses a feasible solution of the COP, found by any exact method, to further decompose the original problem into a bounded number of subproblems which are considerably smaller in size. The number of subproblems is bounded and is controlled by parameter $p$ whereas their size is controlled by depth which is a depth limit after which we stop the decomposition process. The proposed algorithm is designed so (i) to speed up the time to finding the optimal solution by problem decomposition and visiting of promising subproblems first; and (ii) to speed up the proof of optimality by strengthening the cost-based filtering. Experiments on two benchmarks show the efficiency of this algorithm

### 6.2 Discussion and Future work

The findings discussed in this thesis, sparked new research lines for the future. We can identify at least three research lines which look promising for extending our research.

A first research direction is to use the set of he transformations, presented in the second chapter, between models with existed algorithms of knapsack problems to solve other problems. In fact, these transformations may very well prove to be useful in using algorithms (heuristics as a case) already developed. Such as many extensions, the MKP as example, is extensively studied, so it is very interesting to use MKP heuristics to solve the GUBMKP and MMKP.
1. A second line of research, concerns to embedded the objective function into the filtering algorithm of \textit{mcmdk}. Such as, in this thesis we have treated the defined global constraint without the objective function, so we intend to apply the proposed filtering algorithm on challenging real world problems when the problem involve an objective to maximize such as the objective function plays a significant role to reduce the search space.

2. Another promising research direction is to exploit the strength of the proposed algorithm \textit{S&D}. Several lines can be distinguished:

- Test the efficiency and the effect of \textit{S&D} on a large optimization problems.
- Test the effect of \textit{S&D} on the integer linear programs.
- Such as Operation Research techniques have shown Build a hybrid algorithm between hybrid models
Bibliography


