Republic of Tunisia
Ministry of Higher Education and Scientific Research
University of Sfax
National Engineering School of Sfax

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

List of Algorithms

List of Figures

List of Tables

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 constraints. 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 abstractions 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 corresponding *filtering algorithm*, or propagator, in the solver. The filtering algorithm associated to a constraint asserts that the latter must be satisfied and preemptively removes values that would violate it. In simpler terms, a filtering algorithm takes domains as input and produces smaller domains as output by removing values 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 underlying 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 efficient 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 Knapsack Problems involving dimensions, demands and multiple choice constraints: generalization and transformations between formulations *International Journal of Advanced Science and Technology*,2012, 46, 71-94.

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 4

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 *mcmdk* constraint via the proposed filtering algorithm is effective and efficient.

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 *S&D* for short). The proposed strategy is a systematic iterative depth-first strategy that is based on problem decomposition. *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 *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:

A. Lamine and M. Khemakhem and B. Hnich and H. Chabchoub Solving Constrained Optimization Problems by Solution-based Decomposition Search *Journal of Combinatorial Optimization*. 2015

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

$$\textit{Constraint Programming} = \textit{Model} + \textit{Propagation} + \textit{Search}$$

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 *s*olution 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:

$x_1 \neq x_2$

$x_2 \neq 2$

$x_1 + x_2 + x_3 \geq 6$

Decision variables and domains:

$x1 \in \{1, 2\}$

$x2 \in \{1, 2\}$

$x_3 \in \{1, 2, 3, 4, 5\}$

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,

```
1  begin
2  |  D ← propagate(D);
3  |  if D is a false domain then
4  |  |  return;
5  |  end
6  |  if ∃x ∈ X : |D(x)| > 1 then
7  |  |  x ← chooseVariable(X);
8  |  |  D_s ← splitDomain(D(x)));
9  |  |  constraintSearch(D ∪ {x ↦ D_s});
10 |  |  constraintSearch(D ∪ {x ↦ D(x)\D_s)};
11 |  else
12 |  |  Output solution;
13 |  end
14 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 *chooseVariable* (line 7) and value ordering heuristics *splitDomain* (line 8) . These heuristics impose an ordering on the variables and values, respectively. The ordering imposed by these heuristics has a great impact on the search process Rossi et al. (2006b).

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 constraint that captures a relation between a non-fixed number of variables. In general, 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):

- Expressiveness: it is more convenient to define one constraint corresponding to a set of constraints than to define independently each constraint of this set.

- 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 *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, " *Two sided knapsack constraint*", 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.

Two sided knapsack constraint

Trick Trick (2003a) proposed a variation of *subset-sum constraint*, special case of a *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 *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 $X = \{x_1, ..., x_n\}$ 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

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

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

$$\begin{aligned} &\textit{two sided knapsack constraint}(x_1, \ldots, x_n, z, w) = \\ &\{(d_1, \ldots, d_n, d) | \forall i \ d_i \in D(x_i), d \in D(z), d \leq \textstyle\sum_{i=1}^{n} w_i d_i\} \\ &\qquad\qquad \cap \\ &\{(d_1, \ldots, d_n, d) | \forall i \ d_i \in D(x_i), d \in D(z), \textstyle\sum_{i=1}^{n} w_i d_i \leq d\}, \end{aligned}$$

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 (COP)*, also called *constraint satisfaction optimization problems (CSOP)*Tsang (1993), is a CSP P defined on the variables $x_1, \ldots, x_n$ and augmented with an *objective function* $f : D(x_1) \times \cdots \times D(x_n) \rightarrow Q$ that maps the variables $x_1, \ldots, x_n$ to an evaluation score.

An *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 *maximize* z or *minimize* z for a maximization or a minimization problem, respectively called *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. Morever, 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)

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.

FIGURE 2.2: A general form of an integer linear program

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

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

	Type of constraint			
Problem	knapsack	multidimensional knapsack	multiple demand	multiple choice
KP	X			
MKP		X		
MDMKP		X	X	
MCKP	X			X
MMKP		X		X
GUBMKP		X		X
MDMMKP		X	X	X

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_j^k	the weight of item j in knapsack k
c^k	the capacity of knapsack k
q	the number of demand constraints
n	the number of groups
$G = \{G_1, ..., G_n\}$	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.

FIGURE 3.2: An integer linear program of the knapsack problem

$$\begin{aligned} &\text{maximize} && \textstyle\sum_{j=1}^{N} p_j x_j && && (1)\\ &\text{subject to} \\ & && \textstyle\sum_{j=1}^{N} w_j x_j \leq c && && (2)\\ & && x_j \in \{0,1\} && (j = 1,\ldots,N) && (3) \end{aligned}$$

FIGURE 3.3: An instance example of the knapsack problem.

Input:

$N \leftarrow 8$

			$\rightarrow \mathcal{P}rofits \rightarrow$					
p_j	10	20	30	40	50	60	70	80
			$\rightarrow \mathcal{W}eights \rightarrow$					
w_j	5	20	25	35	40	45	55	60
			$\rightarrow \mathcal{C}apacity \rightarrow$					
c			150					

Constraints:

$5x_1 + 20x_2 + 25x_3 + 35x_4 + 40x_5 + 45x_6 + 55x_7 + 60x_8 \leq 150$

$x_j \in \{0,1\} \quad (j = 1,\ldots,8)$

Objective:

$maximize \quad 10x_1 + 20x_2 + 30x_3 + 40x_4 + 50x_5 + 60x_6 + 70x_7 + 80x_8$

Optimal solution:

x_j	1	0	0	0	1	1	0	1	value	200

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_j^k. 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

$$\begin{aligned} &\text{maximize} && \textstyle\sum_{j=1}^{N} p_j x_j && && (4)\\ &\text{subject to} \\ & && \textstyle\sum_{j=1}^{N} w_j^k x_j \leq c^k && (k = 1, \ldots, m) && (5)\\ & && x_j \in \{0, 1\} && (j = 1, \ldots, N) && (6) \end{aligned}$$

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:

$N \leftarrow 8 \quad m \leftarrow 2$

			$\rightarrow \mathcal{P}rofits \rightarrow$					
p_j	10	20	30	40	50	60	70	80
			$\rightarrow \mathcal{W}eights \rightarrow$					
w_j^k	5	20	25	35	40	45	55	60
	90	120	70	110	90	65	80	150
			$\rightarrow \mathcal{C}apacity \rightarrow$					
c^k				150	300			

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:

$maximize \quad 10x_1 + 20x_2 + 30x_3 + 40x_4 + 50x_5 + 60x_6 + 70x_7 + 80x_8$

Optimal solution:

x_j	0	0	1	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

$$\begin{array}{lll}
\text{maximize} & \sum_{j=1}^{N} p_j x_j & (7) \\
\text{subject to} & & \\
 & \sum_{j=1}^{N} w_j^k x_j \leq c^k \quad (k = 1, \ldots, m) & (8) \\
 & \sum_{j=1}^{N} w_j^k x_j \geq c^k \quad (k = 1 + m, \ldots, (m + q)) & (9) \\
 & x_j \in \{0, 1\} \quad (j = 1, \ldots, N) & (10)
\end{array}$$

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

Input:

$N \leftarrow 8 \quad m \leftarrow 2 \quad q \leftarrow 2$

			$\rightarrow \mathcal{P}rofits \rightarrow$					
p_j	10	20	30	40	50	60	70	80
			$\rightarrow \mathcal{W}eights \rightarrow$					
w_j^k	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
			$\rightarrow \mathcal{C}apacity \rightarrow$					
c^k			150	300	80	200		

Constraints:

$5x_1 + 20x_2 + 25x_3 + 35x_4 + 40x_5 + 45x_6 + 55x_7 + 60x_8 \leq 150$
$90x_1 + 120x_2 + 70x_3 + 110x_4 + 90x_5 + 65x_6 + 80x_7 + 150x_8 \leq 300$
$5x_1 + 20x_2 + 100x_3 + 35x_4 + 60x_5 + 45x_6 + 50x_7 + 60x_8 \geq 80$
$90x_1 + 60x_2 + 70x_3 + 110x_4 + 90x_5 + 45x_6 + 20x_7 + 10x_8 \geq 200$
$x_j \in \{0, 1\} \quad (j = 1, \ldots, 8)$

Objective:

$maximize \quad 10x_1 + 20x_2 + 30x_3 + 40x_4 + 50x_5 + 60x_6 + 70x_7 + 80x_8$

Optimal solution:

x_j	0	0	0	1	1	0	1	0	value	160

3.3.4 The multiple choice knapsack problem

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

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

$$\begin{array}{lll}
\text{maximize} & \sum_{i=1}^{n} \sum_{j=1}^{|G_i|} p_{ij} x_{ij} & (12) \\
\text{subject to} & & \\
& \sum_{i=1}^{n} \sum_{j=1}^{|G_i|} w_{ij} x_{ij} \leq c & (13) \\
& \sum_{j=1}^{|G_i|} x_{ij} = 1 \qquad (i = 1, \ldots, n) & (14) \\
& x_{ij} \in \{0, 1\} \qquad (i = 1, \ldots, n), (j = 1, \ldots, |G_i|) & (15)
\end{array}$$

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 a 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:

$N \leftarrow 8 \quad n \leftarrow 3$

$$\rightarrow \mathcal{P}rofits \rightarrow$$

$$p_{ij} \quad \underbrace{10\ 20\ 30}_{G_1} \quad \underbrace{40\ 50}_{G_2} \quad \underbrace{60\ 70\ 80}_{G_3}$$

$$\rightarrow \mathcal{W}eights \rightarrow$$

$$w_{ij} \quad \underbrace{5\ 20\ 25}_{G_1} \quad \underbrace{35\ 40}_{G_2} \quad \underbrace{45\ 55\ 60}_{G_3}$$

$$\rightarrow \mathcal{C}apacities \rightarrow$$

$$c \qquad 130$$

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:

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

Optimal solution:

$$\underbrace{0\ \ 1\ \ 0}_{G_1} \quad \underbrace{0\ \ 1}_{G_2} \quad \underbrace{0\ \ 0\ \ 1}_{G_3} \qquad value \quad 150$$

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

$$\text{maximize} \quad \sum_{i=1}^{n} \sum_{j=1}^{|G_i|} p_{ij} x_{ij} \tag{16}$$

subject to

$$\sum_{i=1}^{n} \sum_{j=1}^{|G_i|} w_{ij}^k x_{ij} \leq c^k \quad (k = 1, \ldots, m) \tag{17}$$
$$\sum_{j=1}^{|G_i|} x_{ij} = 1 \quad (i = 1, \ldots, n) \tag{18}$$
$$x_{ij} \in \{0, 1\} \quad (i = 1, \ldots, n), (j = 1, \ldots, |G_i|) \tag{19}$$

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.

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:

$N \leftarrow 8 \quad n \leftarrow 3 \quad m \leftarrow 2$

$$\rightarrow \mathcal{P}rofits \rightarrow$$

$$p_{ij} \quad \underbrace{10\ 20\ 30}_{G_1} \quad \underbrace{40\ 50}_{G_2} \quad \underbrace{60\ 70\ 80}_{G_3}$$

$$\rightarrow \mathcal{W}eights \rightarrow$$

$$w_{ij} \quad 5 \quad 20 \quad 25 \quad 35 \quad 40 \quad 45 \quad 55 \quad 60$$

$$\underbrace{90\ 120\ 70}_{G_1} \quad \underbrace{110\ 90}_{G_2} \quad \underbrace{65\ 80\ 150}_{G_3}$$

$$\rightarrow \mathcal{C}apacities \rightarrow$$

$$c^k \quad 130\ 230$$

Constraints:

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

Objective:

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

Optimal solution:

$$\underbrace{0\ 0\ 1}_{G_1} \quad \underbrace{0\ 1}_{G_2} \quad \underbrace{1\ 0\ 0}_{G_3} \quad value \quad 140$$

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

$$\text{maximize} \quad \sum_{i=1}^{n} \sum_{j=1}^{|G_i|} p_{ij} x_{ij} \tag{20}$$

subject to

$$\sum_{i=1}^{n} \sum_{j=1}^{|G_i|} w_{ij}^{k} x_{ij} \leq c^{k} \quad (k = 1, \ldots, m) \tag{21}$$

$$\sum_{j=1}^{|G_i|} x_{ij} \leq 1 \quad (i = 1, \ldots, n) \tag{22}$$

$$x_{ij} \in \{0, 1\} \quad (i = 1, \ldots, n), (j = 1, \ldots, |G_i|) \tag{23}$$

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:

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

Objective:

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

Optimal solution:

$$\underbrace{0\ \ 0\ \ 1}_{G_1} \qquad \underbrace{0\ \ 1}_{G_2} \qquad \underbrace{1\ \ 0\ \ 0}_{G_3} \qquad value \quad 140$$

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

$$\text{maximize} \quad \sum_{i=1}^{n} \sum_{j=1}^{|G_i|} p_{ij} x_{ij} \tag{24}$$

subject to

$$\sum_{i=1}^{n} \sum_{j=1}^{|G_i|} w_{ij}^k x_{ij} \leq c^k \quad (k = 1, \ldots, m) \tag{25}$$
$$\sum_{i=1}^{n} \sum_{j=1}^{|G_i|} w_{ij}^k x_j \geq c^k \quad (k = 1 + m, \ldots, (m + q)) \tag{26}$$
$$\sum_{j=1}^{|G_i|} x_{ij} = 1 \quad (i = 1, \ldots, n) \tag{27}$$
$$x_{ij} \in \{0, 1\} \quad (i = 1, \ldots, n), (j = 1, \ldots, |G_i|) \tag{28}$$

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.

FIGURE 3.15: An instance example of the multidimensional multiple demand multiple choice knapsack problem

Input:

$N \leftarrow 8 \quad n \leftarrow 3 \quad m \leftarrow 2$

$$\rightarrow \mathcal{P}rofits \rightarrow$$

$$p_{ij} \quad \underbrace{10\ 20\ 30}_{G_1} \quad \underbrace{40\ 50}_{G_2} \quad \underbrace{60\ 70\ 80}_{G_3}$$

$$\rightarrow \mathcal{W}eights \rightarrow$$

$$w_{ij} \quad \begin{matrix} 5 & 20 & 25 & 35 & 40 & 45 & 55 & 60 \\ 90 & 120 & 70 & 110 & 90 & 65 & 80 & 150 \\ 5 & 20 & 100 & 35 & 60 & 45 & 50 & 60 \end{matrix}$$

$$\underbrace{90\ 60\ 70}_{G_1} \quad \underbrace{110\ 90}_{G_2} \quad \underbrace{45\ 20\ 10}_{G_3}$$

$$\rightarrow \mathcal{C}apacities \rightarrow$$

$$c^k \quad 130\ 230\ 80\ 200$$

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:

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

Optimal solution:

$$\underbrace{0\ 0\ 1}_{G_1} \quad \underbrace{0\ 1}_{G_2} \quad \underbrace{1\ 0\ 0}_{G_3} \quad value \quad 140$$

3.3.8 Relation schema between problems

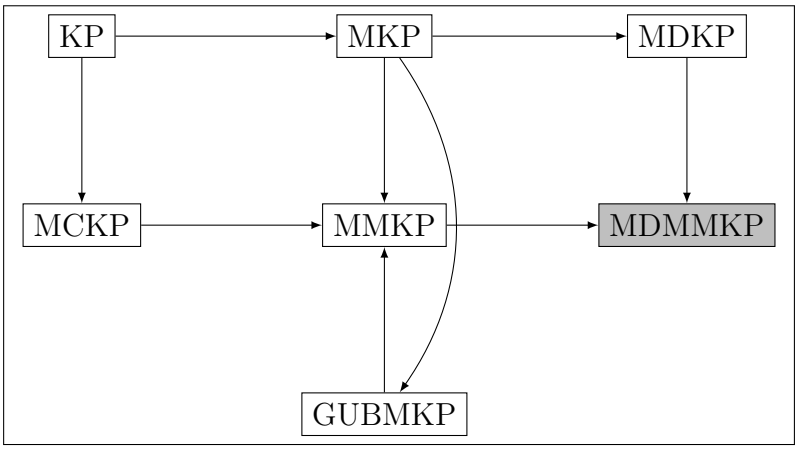
Figure 3.16 shows a set of possible generalizations between the problems mentioned above where each arrow represents a generalization between a target problem and a destination problem. In fact:

- The problems in which knapsack constraint is multidimensional are a generalization of problems with one dimension. Among these problems the MKP

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

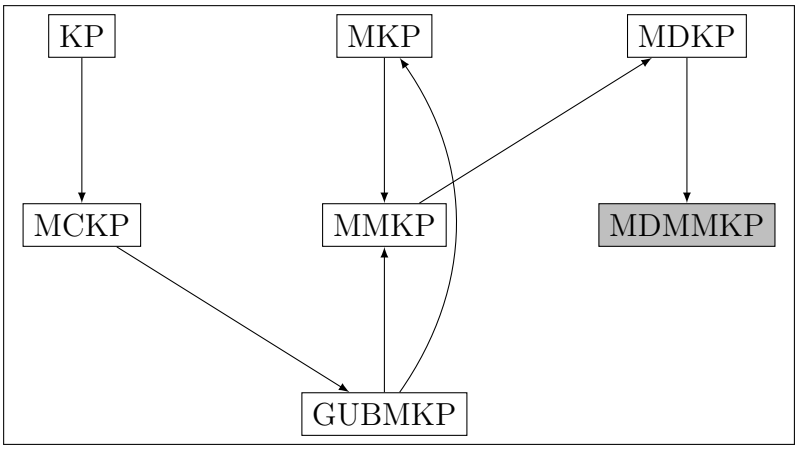


According to the transitivity characteristics of the generalization between problems we consider the MDMMKP as the most generalized problem.

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

FIGURE 3.17: Transformation between ILPs of KPs



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 ($\leq$) 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

$$\text{maximize} \quad \sum_{i=1}^{n} \sum_{j=1}^{|G_i|+1} p_{ij} x_{ij} \tag{29}$$

subject to

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

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

$$x_{ij} \in \{0, 1\} \quad (i = 1, \ldots, n), (j = 1, \ldots, |G_i| + 1) \tag{32}$$

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

Input:

$N \leftarrow 11 \quad n \leftarrow 3 \quad m \leftarrow 2$

$\rightarrow \mathcal{P}rofits \rightarrow$

p_{ij} $\underbrace{10\ \ 20\ \ 30\ \ 0}_{G_1}$ $\underbrace{40\ \ 50\ \ 0}_{G_2}$ $\underbrace{60\ \ 70\ \ 80\ \ 0}_{G_3}$

$\rightarrow \mathcal{W}eights \rightarrow$

w_{ij} 5 20 25 0 35 40 0 45 55 60 0

$\underbrace{90\ \ 120\ \ 70\ \ 0}_{G_1}$ $\underbrace{110\ \ 90\ 0}_{G_2}$ $\underbrace{65\ \ 80\ \ 150\ 0}_{G_3}$

$\rightarrow \mathcal{C}apacities \rightarrow$

c^k 130 230

Constraints:

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

$90x_{11} + 120x_{12} + 70x_{13} + 110x_{21} + 90x_{22} + 65x_{31} + 80x_{32} + 150x_{33} \leq 230$

$x_{11} + x_{12} + x_{13} + x_{14} = 1$

$x_{21} + x_{22} + x_{23} = 1$

$x_{31} + x_{32} + x_{33} + x_{34} = 1$

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

Objective:

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

Optimal solution:

$\underbrace{0\ \ 0\ \ 1\ \ 0}_{G_1}$ $\underbrace{0\ \ 1\ \ 0}_{G_2}$ $\underbrace{1\ \ 0\ \ 0\ \ 0}_{G_3}$ $value \quad 140$

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

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

$$p_{j1} = p_j,\ p_{j2} = 0,\ w^k_{j1} = w^k_j,\ w^k_{j2} = 0\ (j = 1, \ldots, n)\ (k = 1, \ldots, m)$$

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

maximize	$\sum_{i=1}^{N} \sum_{j=1}^{2} p_{ij} x_{ij}$		(33)
subject to			
	$\sum_{i=1}^{N} \sum_{j=1}^{2} w^k_{ij} x_{ij} \leq c^k$	$((k = 1, \ldots, m)$	(34)
	$\sum_{j=1}^{2} x_{ij} = 1$	$(i = 1, \ldots, N)$	(35)
	$x_{ij} \in \{0, 1\}$	$(i = 1, \ldots, N), (j = 1, 2)$	(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 $w^k_{ij} = w^k_l$ 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:

$N \leftarrow 16 \quad m \leftarrow 2$

$$\rightarrow \mathcal{P}rofits \rightarrow$$

$$p_j \quad \underbrace{10\ 0}_{G_1} \quad \underbrace{20\ 0}_{G_2} \quad \underbrace{30\ 0}_{G_3} \quad \underbrace{40\ 0}_{G_4} \quad \underbrace{50\ 0}_{G_5} \quad \underbrace{60\ 0}_{G_6} \quad \underbrace{70\ 0}_{G_7} \quad \underbrace{80\ 0}_{G_8}$$

$$\rightarrow \mathcal{W}eights \rightarrow$$

$$w_j^k \quad 5\ \ 0 \quad 20\ \ 0 \quad 25\ \ 0 \quad 35\ \ 0 \quad 40\ \ 0 \quad 45\ \ 0 \quad 55\ \ 0 \quad 60\ \ 0$$

$$\underbrace{90\ 0}_{G_1} \quad \underbrace{120\ 0}_{G_2} \quad \underbrace{70\ 0}_{G_3} \quad \underbrace{110\ 0}_{G_4} \quad \underbrace{90\ 0}_{G_5} \quad \underbrace{65\ 0}_{G_6} \quad \underbrace{80\ 0}_{G_7} \quad \underbrace{150\ 0}_{G_8}$$

$$\rightarrow \mathcal{C}apacity \rightarrow$$

$$c^k \qquad 150\ 300$$

Constraints:

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

Objective:

$maximize \quad 10x_{11} + 20x_{21} + 30x_{31} + 40x_{41} + 50x_{51} + 60x_{61} + 70x_{71} + 80x_{81}$

Optimal solution:

$$\underbrace{0\ 1}_{G_1} \quad \underbrace{0\ 1}_{G_2} \quad \underbrace{1\ 0}_{G_3} \quad \underbrace{0\ 1}_{G_4} \quad \underbrace{0\ 1}_{G_5} \quad \underbrace{0\ 1}_{G_6} \quad \underbrace{1\ 0}_{G_7} \quad \underbrace{1\ 0}_{G_8} \quad value \quad 180$$

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 $\sum_{l=1}^{N} w_l^k y_l \leq c^k, \forall k \in (m+1, \ldots, m+n)$ where $c^k = 1 \ \forall k \in (m+1, \ldots, m+n)$ and for each $k+m$ knapsack constraint $\forall k \in (1, \ldots, n)$

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

$$
\begin{array}{lll}
\text{maximize} & \sum_{l=1}^{N} p_l y_l & (37) \\
\text{subject to} & & \\
 & \sum_{l=1}^{N} w_l^k y_l \leq c^k \quad (k = 1, \ldots, m+n) & (38) \\
 & y_l \in \{0,1\} \quad (\ (l = 1, \ldots, N) & (39)
\end{array}
$$

$$
w_l^{k+m} = \left\{ \begin{array}{l} 1 \ , \forall l \in (1 + N_{k-1}, \ldots, N_k) \\ 0, \text{ otherwise.} \end{array} \right\}
$$

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

Input:

$N \leftarrow 8 \quad m \leftarrow 5$

$\rightarrow \mathcal{P}rofits \rightarrow$

p_j	10	20	30	40	50	60	70	80

$\rightarrow \mathcal{W}eights \rightarrow$

w_j^k	5	20	25	35	40	45	55	60
	90	120	70	110	90	65	80	150
	1	1	1	0	0	0	0	0
	0	0	0	1	1	0	0	0
	0	0	0	0	0	1	1	1

$\rightarrow \mathcal{C}apacity \rightarrow$

c^k	150	300	1	1	1

Constraints:

$5x_1 + 20x_2 + 25x_3 + 35x_4 + 40x_5 + 45x_6 + 55x_7 + 60x_8 \leq 150$

$90x_1 + 120x_2 + 70x_3 + 110x_4 + 90x_5 + 65x_6 + 80x_7 + 150x_8 \leq 300$

$x_1 + x_2 + x_3 \leq 1$

$x_4 + x_5 \leq 1$

$x_6 + x_7 + x_8 \leq 1$

$x_j \in \{0, 1\} \quad (j = 1, \ldots, 8)$

Objective:

$maximize \quad 10x_1 + 20x_2 + 30x_3 + 40x_4 + 50x_5 + 60x_6 + 70x_7 + 80x_8$

Optimal solution:

x_j	0	0	1	0	1	1	0	0	value	140

where x_{ij^*} is the variable with $w_{ij^*} = min\{w_{ij}^1 \; \forall j \in (1, \ldots, |G_i|)\}, \forall i \; \in (1, \ldots, n)$. For purposes of explanation, we modify the index of items in order that the index j^* of each group G_i is equal to $= |G_i| \quad \forall i \in (1, \ldots, n)$. 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, $c' = c - \sum_{i=1}^{n} w_{ij^*}^1$, $p'_{ij} = p_{ij} - p_{ij^*}$ and $w'_{ij} = w_{ij}^1 - w_{ij^*}^1$.

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

$$\begin{array}{lll}
\text{maximize} & \sum_{i=1}^{n}\sum_{j=1}^{|G_i|} p_{ij}x_{ij} & (40)\\
\text{subject to} & & \\
& \sum_{i=1}^{n}\sum_{j=1}^{|G_i|} w_{ij}^{k}x_{ij} \leq c^{k} \quad ((k=1,\ldots,m) & (41)\\
& \sum_{j=1}^{|G_i|} x_{ij} \leq 1 \quad (i=1,\ldots,n) & (42)\\
& \sum_{j=1}^{|G_i|} x_{ij} \geq 1 \quad (i=1,\ldots,n) & (43)\\
& x_{ij} \in \{0,1\} \quad (i=1,\ldots,n),(j=1,\ldots,|G_i|) & (44)
\end{array}$$

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

$$\begin{array}{lll}
\text{maximize} & \sum_{l=1}^{N} p_l y_l & (45)\\
\text{subject to} & & \\
& \sum_{l=1}^{N} w_l^{k} y_l \leq c^{k} \quad (k=1,\ldots,m+n) & (46)\\
& \sum_{l=1}^{N} w_l^{k} y_l \geq c^{k}, \quad (k=m+n+1,\ldots,m+2n) & (47)\\
& y_l \in \{0,1\} \quad (l=1,\ldots,N) & (48)
\end{array}$$

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.

FIGURE 3.26: An instance example of the MMKP based on the MDMKP formulation

Input:

$N \leftarrow 8 \quad m \leftarrow 5 \quad q \leftarrow 3$

$\rightarrow \mathcal{P}rofits \rightarrow$

p_j	10	20	30	40	50	60	70	80

$\rightarrow \mathcal{W}eights \rightarrow$

w_j^k	5	20	25	35	40	45	55	60
	90	120	70	110	90	65	80	150
	1	1	1	0	0	0	0	0
	0	0	0	1	1	0	0	0
	0	0	0	0	0	1	1	1
	1	1	1	0	0	0	0	0
	0	0	0	1	1	0	0	0
	0	0	0	0	0	1	1	1

$\rightarrow \mathcal{C}apacity \rightarrow$

c^k	150	300	1	1	1	1	1	1

Constraints:

$5x_1 + 20x_2 + 25x_3 + 35x_4 + 40x_5 + 45x_6 + 55x_7 + 60x_8 \leq 150$

$90x_1 + 120x_2 + 70x_3 + 110x_4 + 90x_5 + 65x_6 + 80x_7 + 150x_8 \leq 300$

$x_1 + x_2 + x_3 \leq 1$

$x_4 + x_5 \leq 1$

$x_6 + x_7 + x_8 \leq 1$

$x_1 + x_2 + x_3 \geq 1$

$x_4 + x_5 \geq 1$

$x_6 + x_7 + x_8 \geq 1$

$x_j \in \{0, 1\} \quad (j = 1, \ldots, 8)$

Objective:

$maximize \quad 10x_1 + 20x_2 + 30x_3 + 40x_4 + 50x_5 + 60x_6 + 70x_7 + 80x_8$

Optimal solution:

x_j	0	0	1	0	1	1	0	0	value	140

3.5 Experimental results

This section is aimed at:

- validating experimentally the transformations between problems.

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

$$\text{maximize} \quad \bar{v} + \sum_{i=1}^{n} \sum_{j=1}^{|G_i|-1} p'_{ij} x_{ij} \tag{49}$$

subject to

$$\sum_{i=1}^{n} \sum_{j=1}^{|G_i|-1} w'_{ij} x_{ij} \leq c' \tag{50}$$

$$\sum_{j=1}^{|G_i|-1} x_{ij} \leq 1 \qquad (i = 1, \ldots, n) \tag{51}$$

$$x_{ij} \in \{0, 1\} \ (i = 1, \ldots, n) \qquad (j = 1, \ldots, |G_i| - 1) \tag{52}$$

- 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

Input:

$N \leftarrow 8 \quad n \leftarrow 3$

$$\rightarrow \mathcal{P}rofits \rightarrow$$

$$p_{ij} \quad \underbrace{10\ 20}_{G_1} \quad \underbrace{10}_{G_2} \quad \underbrace{10\ 20}_{G_3}$$

$$\rightarrow \mathcal{W}eights \rightarrow$$

$$w_{ij} \quad \underbrace{15\ 20}_{G_1} \quad \underbrace{5}_{G_2} \quad \underbrace{10\ 15}_{G_3}$$

$$\rightarrow \mathcal{C}apacities \rightarrow$$

$$c \quad 45 = 130 - (5 + 35 + 45)$$

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:

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

Optimal solution:

$$\underbrace{1\ 0}_{G_1} \quad \underbrace{1}_{G_2} \quad \underbrace{0\ 1}_{G_3} \quad value \quad 150$$

TABLE 3.3: *Test MKP instances details*

Set	N	m_1
mknapcb1	100	5
mknapcb2	250	5
mknapcb3	500	5
mknapcb4	100	10
mknapcb5	250	10
mknapcb6	500	10

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.

TABLE 3.4: *Test MDMKP instances details*

Set	N	m_1	m_2
mdmkp_ct1	100	5	1 2 5
mdmkp_ct2	250	5	1 2 5
mdmkp_ct3	500	5	1 2 10
mdmkp_ct4	100	10	1 5 10
mdmkp_ct5	250	10	1 5 10
mdmkp_ct6	500	10	1 5 10

For the MMKP, the instances set is summarized in Table 3.5. The header Instance, n, n_i $\sum n_i$ 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 $I01, \ldots, I13$) varying from small to large-scale size ones. These instances are given by Khan et al. (2002).

TABLE 3.5: *Test MMKP instances details*

Instance	n	n_i	$\sum n_i$	m_1
I01	5	5	25	5
I02	10	5	50	5
I03	15	10	150	10
I04	20	10	200	10
I05	25	10	250	10
I06	30	10	300	10
I07	100	10	1000	10
I08	150	10	1500	10
I09	200	10	2000	10
I10	250	10	2500	10
I11	300	10	3000	10
I12	350	10	3500	10
I13	400	10	4000	10

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*

Set	n	n_i	$\sum n_i$	m_1
gubmkp1	10	250	2500	5
gubmkp2	10	500	5000	5
gubmkp3	10	1000	10000	5
gubmkp4	10	1500	15000	5

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*

Set	n	n_i	$\sum n_i$
mckp1	1000	1000	1000000
mckp2	1500	1000	1500000
mckp3	2000	1000	2000000

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 $Dev = \frac{T2}{T1} \times 100$. *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*

Set	m_{T1}	m_{T2}	*Dev*
GUBMKP→ MMKP			
gubmkp1	1,8	1,3	71,2%
gubmkp2	2,8	2,3	82,6%
gubmkp3	5,0	4,3	84,7%
gubmkp4	8,4	8,2	98,0%
MKP→ MMKP			
mknapcb1	3,0	2,9	98,7%
mknapcb2	129,0	126,1	97,7%
mknapcb3	1743,8	1753,2	100,5%
mknapcb4	23,9	24,0	100,1%
mknapcb5	3088,2	3078,6	99,7%
mknapcb6	1952,1	1958,8	100,3%
MDMKP→ MDMMKP			
mdmkp_ct1	48,3	48,2	99,9%
mdmkp_ct2	2437,4	2431,9	99,8%
mdmkp_ct3	2394,0	1939,1	81,0%
mdmkp_ct4	1414,5	1408,3	99,6%
mdmkp_ct5	2509,8	2501,5	99,7%
mdmkp_ct6	3282,7	3238,4	98,6%
MMKP→ MDMKP			
I1	0	0	–
I2	0	0	–
I3	2	2	100,0%
I4	54	26	48,1%
I5	0	0	–
I6	0	0	–
I7	4771	4739	99,3%
I8	6052	4933	81,5%
I9	4735	4287	90,5%
I10	5411	4736	87,5%
I11	4517	4544	100,6%
I12	5494	4904	89,3%
I13	6700	5096	76,1%
MCKP→ GUBMKP			
mckp1	21,8	21,0	96,0%
mckp2	43,5	32,2	73,9%
mckp3	44,8	43,5	97,1%

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 Rgin (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 *kp*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 *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:

$$
mcmdk \begin{cases}
\sum_{i=1}^{n} y_{k,i} \leq c_k & \forall k \in 1, \ldots, m & (1) \\
(x_i = j) \Rightarrow (y_{k,i} = w_{k,i,j} \forall k \in 1, \ldots, m) & \forall i \in 1, \ldots, n, \forall j \in 1, \ldots, |S_i| & (2) \\
x_i \in \{1, \ldots, |S_i|\} & \forall i \in 1, \ldots, n & (3) \\
y_{k,i} = \{w_{k,i,j} | \forall j \in 1, \ldots, |S_i|\} & \forall i \in 1, \ldots, n, \forall k \in 1, \ldots, m & (4)
\end{cases}
$$

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 *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 $j \in \cup_{i=1,\ldots,n} D(x_i)$ be integers and let c be an array of integers, i.e. $c = [c_1, c_2, \ldots, c_m]$. Then, the *mcmdk* constraint is defined as:

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

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:

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

4.3.1 Fundamental properties

Some classical properties can be applied in the pre-processing phase of the filtering algorithm of *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 *mcmdk* is inconsistent.

Input:

$n \leftarrow 3 \;\; m \leftarrow 2$

$\rightarrow \mathcal{W}eights \rightarrow$

$w_{1,i,j}$ 4 2 9 11 3 11 0 5 7 1 6 10

$w_{2,i,j}$ $\underbrace{2\;\;4\;\;3\;\;7}_{S_1}$ $\underbrace{1\;\;0\;\;9\;\;6}_{S_2}$ $\underbrace{8\;\;13\;\;10\;\;5}_{S_3}$

$\rightarrow \mathcal{C}apacities \rightarrow$

$c_1 \leftarrow 14 \;\; c_2 \leftarrow 14$

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:

$y_{1,1} = \{2, 4, 9, 11\}$ $y_{1,2} = \{0, 3, 5, 11\}$ $y_{1,3} = \{1, 6, 7, 10\}$

$y_{2,1} = \{2, 3, 4, 7\}$ $y_{2,2} = \{0, 1, 6, 9\}$ $y_{2,3} = \{5, 8, 10, 13\}$

$x_1 = \{1, 2, 3, 4\}$ $x_2 = \{1, 2, 3, 4\}$ $x_3 = \{1, 2, 3, 4\}$

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 $D(y_{k,f}) >= lb$ 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 $D(y_{kf}), k \in 1,\ldots,m$ and j of $D(x_f)$ are also inconsistent (the item j of S_f cannot configurate 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 *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 $D(y_{k_2,f}),\ k_2\ in1,\ldots,m\ k_2 \neq k$.

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

```
1 begin
2 |   for k ← 1 to m do
3 |   |   for i ← 1 to n do
4 |   |   |   put(i, k, Q);
5 |   |   end
6 |   end
7 end
```

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 initialize) and propagation (lines 3-20).

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

Once procedure 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 $getInf(Variable\ y[][], Integer\ k,\ Integer\ f)$ 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 *sum* added to value *val* which corresponds to the maximum value of the variable $y_{k,f}$(value returned by the function

```
1  begin
2      Initialize(Q, n, m);
3      while Q! = ∅ do
4          ⟨k, f⟩ ← get(Q);
5          sum ← getInf(y, k, f);
6          while sum + getSup(y[k][f]) > c[k] do
7              val ← getSup(y[k][f]);
8              remVal(y[k][f], val);
9              j ← getIndex(k, f, val, y);
10             remVal(x_f, j);
11             for k2 ← 1 to m do
12                 if k2 ≠ k then
13                     if w[k2][f][j] = getInf(y[k2][f]) then
14                         putSet(f2, f, k2, Q);
15                     end
16                     remVal(y[k2][f], w[k2][f][j]);
17                 end
18             end
19         end
20     end
21 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.

pair	satisfaction	variables	list Q
$\langle 1,1\rangle$	$(0+1)+11 \leq 14$	–	$\langle 2,3\rangle$ $\langle 2,2\rangle$ $\langle 2,1\rangle$ $\langle 1,3\rangle$ $\langle 1,$
$\langle 1,2\rangle$	$(1+2)+11 \leq 14$	–	$\langle 2,3\rangle$ $\langle 2,2\rangle$ $\langle 2,1\rangle$ $\langle 1,3\rangle$
$\langle 1,3\rangle$	$(2+0)+10 \leq 14$	–	$\langle 2,3\rangle$ $\langle 2,2\rangle$ $\langle 2,1\rangle$
$\langle 2,1\rangle$	$(0+5)+7 \leq 14$	–	$\langle 2,3\rangle$ $\langle 2,2\rangle$
$\langle 2,2\rangle$	$(2+5)+9 > 14$	$y_{2,2} = \{0,1,6,\not{9}\}$ $y_{1,2} = \{\not{0},3,5,11\}$	$\langle 1,3\rangle$ $\langle 1,1\rangle$ $\langle 2,3\rangle$
$\langle 2,3\rangle$	$(2+0)+13 > 14$	$y_{2,3} = \{5,8,10,\not{13}\}$ $y_{1,3} = \{\not{1},6,7,10\}$	$\langle 1,2\rangle$ $\langle 1,3\rangle$, $\langle 1,1\rangle$
$\langle 1,1\rangle$	$(3+6)+11 > 14$	$y_{1,1} = \{2,4,9,\not{11}\}$ $y_{2,1} = \{2,3,4,\not{7}\}$	$\langle 1,2\rangle$ $\langle 1,3\rangle$
	$(3+6)+9 > 14$	$y_{1,1} = \{2,4,\not{9}\}$ $y_{2,1} = \{2,\not{3},4,\}$	
$\langle 1,3\rangle$	$(2+3)+10 > 14$	$y_{1,3} = \{6,7,\not{10}\}$ $y_{2,3} = \{\not{5},8,10\}$	$\langle 2,2\rangle$, $\langle 2,1\rangle$, $\langle 1,2\rangle$
$\langle 1,2\rangle$	$(2+6)+11 > 14$	$y_{1,2} = \{3,5,\not{11}\}$ $y_{2,2} = \{\not{0},1,6\}$	$\langle 2,3\rangle$, $\langle 2,2\rangle$, $\langle 2,1\rangle$
$\langle 2,1\rangle$	$(1+8)+4 \leq 14$	–	$\langle 2,3\rangle$, $\langle 2,2\rangle$
$\langle 2,2\rangle$	$(2+8)+6 > 14$	$y_{2,2} = \{1,\not{6}\}$ $y_{1,2} = \{3,\not{5}\}$	$\langle 2,3\rangle$
$\langle 2,3\rangle$	$(2+1)+10 \leq 14$		$\oslash$

TABLE 4.1: A worked example

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): $Q = \{\langle 2,3\rangle\ \langle 2,2\rangle\ \langle 2,1\rangle\ \langle 1,3\rangle\ \langle 1,2\rangle\ \langle 1,1\rangle\}$.

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 $set_2 =$

$\{(1,5),(5,2),(6,4)\}$.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 configurate 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:

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

We name our filtering algorithm **mcmdk-fal** and the straightforward conjunction **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:

$$c_k = 1/2 \left(\sum_{i=1}^{n} \underbrace{min}_{1 \leq j \leq |S_i|} w_{k,i,j} + \sum_{i=1}^{n} \underbrace{max}_{1 \leq j \leq |S_i|} w_{k,i,j} \right)$$

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.

n	m	**mcmdk-conj**			**mcmdk-fal**		
		CPU	BT	NODE	CPU	BT	NODE
15	10	0	25	60	0	16	29
	12	1	154	700	0	45	290
	14	22	2121	10456	5	464	4360
	16	55	4565	23044	9	709	6723
	18	127	9042	46601	25	1766	17217
	20	1051	64150	340038	406	23161	254600
	22	818	45101	236492	307	16454	172047
	24	1218	61068	328298	330	15579	166786
	26	977	49601	264322	247	11734	120687
	28	1729	80489	427152	495	17639	184167

TABLE 4.2: Comparison on instances with $n = 15$

As predicted by the complexity of *mcmdk*, **mcmdk-fal** and **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 **mcmdk-fal**. We observe that our proposed **mcmdk-fal** introduced in this chapter performs way better than the basic model in all tested instances.

n	m	**mcmdk-conj**			**mcmdk-fal**		
		CPU	BT	NODE	CPU	BT	NODE
10	15	4	410	2111	1	114	1085
12		78	7228	37435	18	1611	15660
14		45	3921	20799	7	626	6237
16		132	11175	57865	30	2503	24452
18		139	11415	57638	25	2019	19980
20		117	9399	46935	24	1947	19264
22		121	10747	44932	15	1150	10118
24		222	16632	81791	34	2565	25191
26		248	18736	85314	74	4681	45121
28		416	28751	141651	75	5006	49105
30		211	15162	70192	33	2408	21634
32		391	30275	127459	119	8961	74969
34		1129	85582	398771	364	23163	229877
36		997	61378	278173	311	19383	191464
38		979	64656	274952	575	34985	196825

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

4.6 Conclusion

In this chapter we have presented *mcmdk*, a new global constraint belonging to the weighted constraints and closely related to the knapsack constraint. We have shown how the *mcmdk* constraint can be modelled using the conjunction of *sum* and *implies* constraints. We have shown that the *mcmdk* constrain is *NP*-complete. Also, we have proposed a simple filtering algorithm to solve it. Our 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

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 an 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 $x_1, ..., x_n$ 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: $left_i$ which contains all values in the domain of y_i less than or equal to $\bar{y}_i$ and $right_i$ which contains all values strictly greater than $\bar{y}_i$. Thus, $left_i$ and $right_i$ form a partition of the domain of y_i.

Using $left_i$, $right_i$, 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 $left_i$ or $right_i$. 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) $y_1 \in \{1, 2\}$, $y_2 \in \{4\}$, and $y_3 \in \{1, 2, 3\}$
- (d) $y_1 \in \{1, 2\}$, $y_2 \in \{4\}$, and $y_3 \in \{4\}$
- (e) $y_1 \in \{3, 4\}$, $y_2 \in \{1, 2, 3\}$, and $y_3 \in \{1, 2, 3\}$
- (f) $y_1 \in \{3, 4\}$, $y_2 \in \{1, 2, 3\}$, and $y_3 \in \{4\}$
- (g) $y_1 \in \{3, 4\}$, $y_2 \in \{4\}$, and $y_3 \in \{1, 2, 3\}$
- (h) $y_1 \in \{3, 4\}$, $y_2 \in \{4\}$, 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 $minmin$. 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 $minmax$. 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 $maxleft$. 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 ($upperB$) and use the solution S to sequentially generate the subproblems in increasing order of promise (using one the heuristics discussed before) and recurse. 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 internal node. Before we try to solve the subproblem, we fail if the current $upperB$ is smaller than a valid lower bound of the subproblem. If, however, the lower bound is smaller than the current $upperB$, we modify our CSP by adding a constraint that states that all solutions must have an objective value smaller than $upperB$. If the subproblem is infeasible, we also fail. When we backtrack to the root node, the value $upperB$ constitutes the value of the optimal solution. Note that if the decomposition process reach a specified depth, we do not decompose the subproblems 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 $depth$ and outputs the optimal solution ($bestSol$) along with its objective function value $upperB$. At line 3, we set the global variable $upperB$ to infinity whereas at line 4 we let pb the CSP corresponding to oPb. The final step, in line 5, computes the $bestSol$ by calling the function $solveAndDecompose$ shown in Algorithm 3.

The function $solveAndDecompose$ has four parameters: the csp pb, the function f, the depth limit $depth$, and the initial 0. At line 2, if the current best $upperB$ is smaller than or equal to a valid lower bound of pb computed by calling function $lowerBound()$, 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 pb a constraint that enforces that any feasible solution shall improve on $upperB$.

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: $min \quad \sum_i y_i$ s.t.

$$y_1 + y_2 \geq 3, y_1 + y_3 \geq 3, y_2 + y_3 \geq 3 \quad where \quad y_i \in \{1, \ldots, 5\}$$

Figure 5.1 shows a trace of the full search tree using the basic $S\&D$ to find the optimal solution ($bestSol = (y_1 = 1, y_2 = 2, y_3 = 2)$). The search tree generated by $S\&D$ is of depth 2. The green nodes present a feasible solution obtained by the function *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 $subPbl1$ according to our heuristic $maxleft$. We find a better feasible solution that improves the objective function value to 5 (which is the optimal solution). We decompose $subPbl1$ using the feasible solution found and start exploring at depth 2 $subPbl11$. However, $subPbl11$ is infeasible because it cannot improve on the value of $upperB$. We backtrack and explore the next most promising subproblem $subPbl12$. But, this time, we immediately fail because the lower bound of $subPbl12$ is 5 which means we cannot improve $upperB$. 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 *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 $\{c_1 \dots c_m\}$ 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 $slab_j$ 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 1,0,0two binary variables $order_{i,j}$ and $color_{i,j}$. $order_{i,j}$ is equal to 1 iff order i is assigned to slab j whereas $color_{i,j}$ 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 [1] 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?

[1] http://www.emn.fr/z-info/choco-solver/

- 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

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.$100 * (optimalsolutionvalue - foundsolutionvalue)/(optimalsolutionvalue)$. We observe that the optimal solution is found in 81.25% for the MMKP instances (respectively 76.7% for the SMSDP insances) 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 Hoeve and Milano (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.

```
input  : oPb = (X, D, C, f); depth : Integer;
output: bestSol : Solution; upperB : Integer;
1  begin
2      upperB is a global variable;
3      upperB ← ∞;
4      pb = (C, D, X);
5      bestSol ← solveAndDecompose(pb, f, depth, 0);
6  end
```

Algorithm 2: Basic *S&D*

```
input  : pb = (X, D, C) : CSP; f : function;
output: bestSol : Solution;
1  begin
2      if upperB > lowerBound(pb) then
3          add f(Y) < upperB to constraints of pb;
4          if levelD ≤ depth then
5              sol ← solve(pb);
6              if sol ≠ null then
7                  upperB ← f(sol);
8                  bestSol ← sol;
9                  while pb has more subproblems do
10                     subPb ← generateNextPromisingSubproblem(pb, sol);
11                     solveAndDecompose(subPb, f, levelID + 1);
12                 end
13             end
14         else
15             sol ← optimize(pb, f);
16             if sol ≠ null then
17                 upperB ← f(sol);
18                 bestSol ← sol;
19             end
20         end
21     end
22     return bestSol;
```

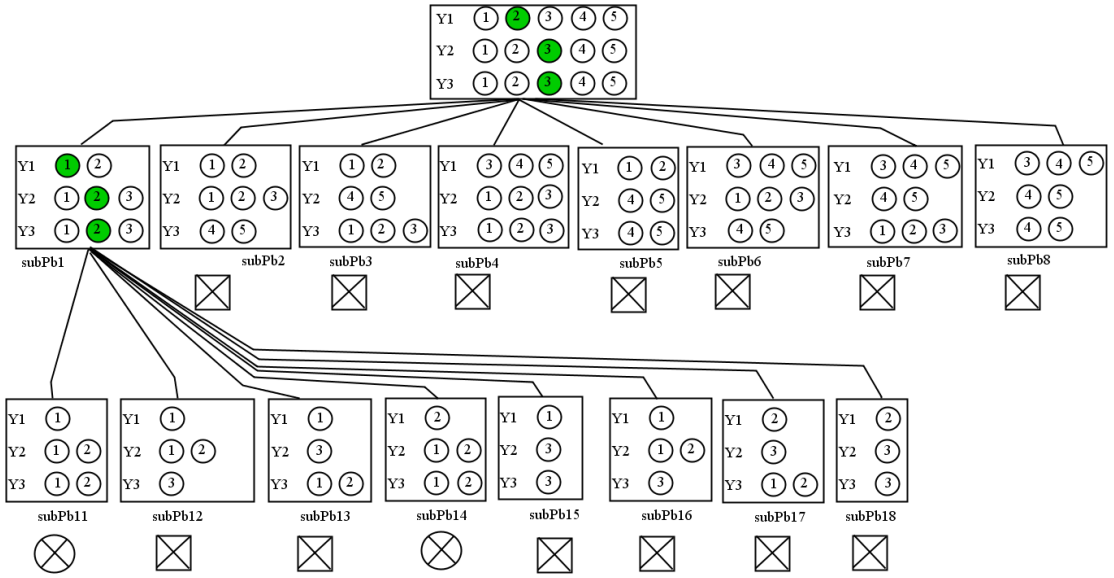


FIGURE 5.1: A Trace of the basic *S&D* on an example.

Constraints:

(1) $\sum_{i=1}^{n} weight_{k,i} \leq c_k \; \forall k \in 1, \ldots, m$

(2) $profit_i = p_{i,item_i} \; \forall i \in \{1, \ldots, n\}$

(3) $weight_{k,i} = w_{k,i,item_i} \; \forall k \in \{1, \ldots, m\}, \forall i \in \{1, \ldots, n\}$

Decision variables and domains:

$profit_i \in \{p_{ij} | \forall j \in (1, \ldots, |G_i|)\}, \;\; \forall i \in \{1, \ldots, n\}$

$item_i \in \{1, \ldots, |G_i|\}, \;\; \forall i \in \{1, \ldots, n\}$

$weight_{k,i} = \{w_{k,i,j} | \forall j \in \{1, \ldots, |G_i|\}\}, \;\; \forall i \in \{1, \ldots, n\}, \forall k \in \{1, \ldots, m\}$

Objective function:

maximize $\sum_{i=1}^{n} profit_i$

FIGURE 5.2: Multidimensional multiple choice knapsack problem formulation.

Constraints:

(1) $\sum_{i=1}^{n} weight(i) * order_{i,j} \leq slab_j, \;\; \forall j \in \{1, \ldots, n\}$

(2) $\sum_{i=1}^{k} color_{i,j} \leq p, \;\; \forall j \in \{1, \ldots, n\}$

(3) $(order_{i,j} = 1) \rightarrow (color_{color(i),j} = 1), \;\; \forall i, j \in \{1, \ldots, n\}$

(4) $slab_j \leq slab_{j+1}, \;\; \forall j \in \{1, ..., n-1\}$

(5) $\sum_{i=1}^{n} weight(i) \leq \sum_{j=1}^{n} slab_j$

Decision variables and domains:

$slab_j \in \{0, c_1, \ldots, c_m\}, \;\; \forall j \in \{1, \ldots, n\}$

$order_{i,j} \in \{0, 1\}, \;\; \forall i, j \in \{1, \ldots, n\}$

$color_{i,j} \in \{0, 1\}, \;\; \forall i \in \{1, \ldots, k\}, \forall j \in \{1, \ldots, k\}$

Objective function:

minimize $\sum_{j=1}^{n} slab_j$

FIGURE 5.3: Steel Mill Slab Design Problem formulation.

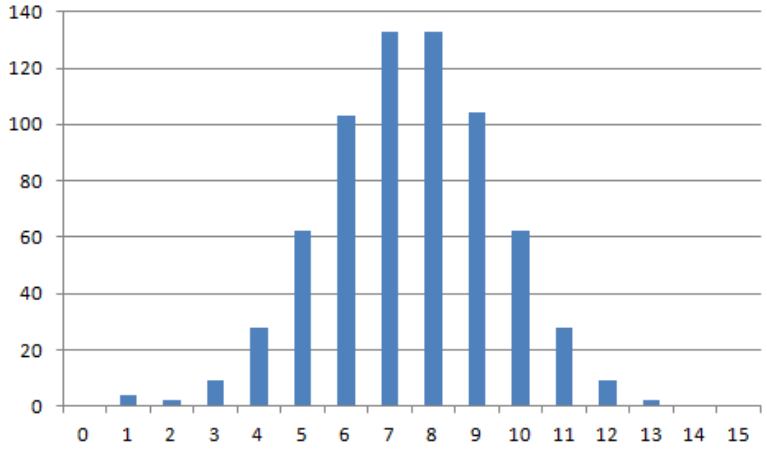


FIGURE 5.4: Time consumed by $S\&D$ in each level for MMKP instance inst17

TABLE 5.1: The runtime in 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.

		Basic *S&D*					Improved *S&D*					*B&B*				GAP	
	$\|y\|$	L_p	Obj	CPU	FindObj	Backtracks	Nodes	L_p	Obj	Obj_{L_p}	CPU	FindObj	Backtracks	Nodes	Obj	CPU	Backtracks	Nodes	*S&D/BS&D*	*S&D/B&B*
Inst1	10	2	946	27	8	16669	2566	1	946	930	8	6	18604	2827	946	36	78274	9727	70,37	77,78
Inst2		0	918	23	3	7215	1147	0	918	918	3	3	7264	1156	918	14	25915	2872	86,96	78,57
Inst3		0	852	48	15	65409	10374	0	852	852	41	15	90761	13516	852	262	518387	53704	14,58	84,35
Inst4		0	924	22	1	6477	1191	0	924	924	3	1	6510	1066	924	20	43926	5437	86,36	85,00
Inst5		0	905	24	3	10319	1721	0	905	905	5	3	11832	1984	905	36	78901	9769	79,17	86,11
Inst6		2	914	22	2	4542	756	1	914	913	2	2	4988	859	914	16	34733	4335	90,91	87,50
Inst7		2	911	24	3	9818	2030	0	911	896	7	1	16421	2822	911	56	111087	12248	70,83	87,50
Inst8		0	885	25	3	11929	2574	0	885	885	8	3	17948	3541	885	68	158237	18545	68,00	88,24
Inst9		0	894	22	2	4621	545	0	894	894	2	2	4744	564	894	27	58545	5967	90,91	92,59
Inst10		0	952	20	0	1122	183	0	952	952	0	0	1221	214	952	13	27052	3497	100,00	100,00
inst11	15	2	1371	800	113	245892	38538	1	1371	1350	132	89	284992	44823	1371	1840	5018578	532953	83,50	92,83
inst12		1	1411	682	2	4993	1258	1	1411	1411	3	2	6681	1703	1411	64	167786	25279	99,56	95,31
inst13		2	1389	700	23	37004	6058	0	1389	1386	22	18	38727	6016	1389	637	1337662	116211	96,86	96,55
inst14		0	1429	685	3	7031	1063	0	1429	1429	3	3	7050	1071	1429	109	236243	26797	99,56	97,25
inst15		1	1408	691	7	15747	2724	0	1408	1394	7	7	15733	2720	1408	301	483579	46325	98,99	97,67
inst16		2	1437	688	6	10337	1634	2	1437	1437	8	6	10359	1644	1437	375	662495	62706	98,84	97,87
inst17		1	1419	685	4	11382	1983	1	1419	1419	6	4	13500	2442	1419	293	608031	63587	99,12	97,95
inst18		1	1418	690	11	23588	3510	1	1418	1418	12	11	24300	3723	1418	626	1393810	130453	98,26	98,08
inst19		3	1421	702	24	38562	8024	2	1421	1420	20	11	43326	8930	1421	1346	4037271	395356	97,15	98,51
inst20		0	1406	684	2	3534	572	0	1406	1406	2	2	3570	585	1406	232	426323	41651	99,71	99,14
inst21	20	1	1873	–	29	81624	18681	1	1873	1873	53	29	130363	30194	1873	259	866599	99032	98,53	79,54
inst22		2	1880	–	68	198591	38605	2	1880	1880	79	68	215183	42653	1880	533	1569832	241784	97,81	85,18
inst23		1	1891	–	18	46093	8920	1	1891	1891	19	18	47405	9212	1891	324	1136532	114932	99,47	94,14
inst24		0	1888	–	88	212369	34315	0	1888	1888	89	89	212606	34405	1888	2519	7312161	616286	97,53	96,47
inst25		0	1908	–	63	171951	34428	0	1908	1908	63	63	172038	34469	1606	–	6971132	797158	98,25	98,25
inst26		0	1909	–	61	165008	20419	0	1909	1909	61	60	165412	20543	1345	–	4547720	480131	98,31	98,31
inst27		0	1873	–	41	107449	16894	0	1873	1873	50	41	122105	20028	1570	–	6520887	652205	98,61	98,61
inst28		1	1934	–	8	18735	3267	1	1934	1934	9	8	18858	3316	1934	964	2812825	234288	99,75	99,07
inst29		1	1925	–	2	7731	2007	1	1925	1925	3	2	7908	2078	1925	396	891547	94241	99,92	99,24
inst30		0	1904	–	4	12605	3238	0	1904	1904	5	4	13672	3516	1904	2988	7888102	747909	99,86	99,83
inst31	25	3	2377	–	63	2874	1145	3	2377	2377	67	63	2880	1148	2377	101	285220	35789	98,14	33,66
inst32		2	2403	–	109	316107	50595	2	2403	2403	111	109	316255	50662	2403	280	1038589	149137	96,92	60,36
inst33		1	2421	–	29	52596	5718	1	2421	2421	30	29	52598	5719	2421	154	603329	78062	99,17	80,52
inst34		3	2326	–	143	353463	46195	2	2326	2319	134	121	365201	49379	2326	1289	3194546	287518	96,28	89,60
inst35		1	2350	–	14	50890	10586	0	2350	2345	14	12	50354	10457	2350	195	674024	83994	99,61	92,82
inst36		4	2402	–	294	66988	11823	3	2402	2398	86	56	68482	12138	2402	1355	5353288	620193	97,61	93,65
inst37		0	2372	–	6	22207	5742	0	2372	2372	7	6	22888	5952	2372	156	526495	88169	99,81	95,51
inst38		3	2399	–	69	102955	15843	3	2399	2399	106	69	103047	15882	2399	3567	9216292	888753	97,06	97,03
inst39		0	2397	–	50	166779	26163	0	2397	2397	52	50	172611	27667	2397	1883	5917178	709551	98,56	97,24
inst40		2	2408	–	13	15932	4493	1	2408	2407	7	4	20759	5567	2408	950	4414479	569512	99,81	99,26

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.

		Basic *S&D*					Improved *S&D*					*B&B*				GAP	
	$\|y\|$	L_p	Obj	CPU	FindObj	Backtracks	Nodes	L_p	Obj	Obj_{L_p}	CPU	FindObj	Backtracks	Nodes	Obj	CPU	Backtracks	Nodes	*S&D/BS&D*	*S&D/B&B*
Inst41	10	0	972	49	29	47400	5767	0	972	972	32	29	55427	7321	972	101	188975	21147	34,69	68,32
Inst42		0	977	22	2	2450	417	0	977	977	3	2	2752	492	977	80	113079	8570	86,36	96,25
Inst43		0	967	22	1	4425	719	0	967	967	2	1	6336	1147	967	70	85474	6061	90,91	97,14
Inst44		0	962	23	2	4259	467	0	962	962	3	2	4946	590	962	42	83333	8276	86,96	92,86
Inst45		1	964	23	3	6714	1010	0	964	961	4	2	7548	1137	964	40	74261	7072	82,61	90,00
Inst46		0	979	20	0	514	135	0	979	979	0	0	513	129	979	59	95449	8958	100,00	100,00
Inst47		0	977	24	3	7515	978	0	977	977	4	3	7804	1063	977	30	64872	9328	83,33	86,67
Inst48		0	975	26	5	11289	1595	0	975	975	6	5	11374	1616	975	42	78550	7940	76,92	85,71
Inst49		0	970	27	7	11257	1215	0	970	970	7	7	11456	1279	970	52	75181	6972	74,07	86,54
Inst50		0	961	26	6	15196	2277	0	961	961	8	6	19958	3258	961	96	14511	6972	69,23	91,67
inst51	15	1	1457	739	51	105227	18493	0	1457	1456	53	50	111879	19660	1457	225	478341	51973	92,83	76,44
inst52		1	1472	763	79	172706	23898	1	1472	1472	80	79	173001	23995	1472	2268	4138219	443219	89,52	96,47
inst53		0	1474	692	3	7283	1423	0	1474	1474	3	3	7347	1445	1474	329	779456	73210	99,57	99,09
inst54		2	1482	694	5	12189	2774	2	1482	1482	7	5	12376	2839	1482	75	160646	20891	98,99	90,67
inst55		0	1471	708	24	49580	7319	0	1471	1471	24	24	50072	7469	1471	780	1828494	226034	96,61	96,92
inst56		0	1461	744	53	162497	28588	0	1461	1461	70	53	204251	36983	1386	–	6785020	621847	90,59	98,06
inst57		1	1467	705	15	46701	8164	1	1467	1467	17	15	48809	8616	1467	540	980759	93465	97,59	96,85
inst58		0	1474	718	28	56794	8841	0	1474	1474	33	28	71787	11626	1474	1233	3469606	451426	95,40	97,32
inst59		0	1452	702	14	27333	4414	0	1452	1452	14	14	27943	4612	1249	–	7432601	751559	98,01	99,61
inst60		1	1479	706	16	42229	7312	0	1479	1475	16	16	42611	7421	1479	615	1652987	220869	97,73	97,40
inst61	20	1	1962	–	32	98721	21914	1	1962	1962	33	32	98860	21960	1962	476	1199474	166108	99,08	93,07
inst62		0	1974	–	119	105688	11510	0	1974	1974	118	118	106223	11697	1220	–	2277978	104397	96,72	96,72
inst63		0	1952	–	18	67134	13286	0	1952	1952	18	18	67472	13414	1952	187	587387	101147	99,50	90,37
inst64		1	1971	–	56	169240	40668	1	1971	1971	58	56	177059	42991	1971	905	2540425	418583	98,39	93,59
inst65		1	1964	–	12	42673	12146	1	1964	1964	12	12	43088	12309	1964	375	1298019	325471	99,67	96,80
inst66		1	1970	–	29	63393	9513	1	1970	1970	29	29	64185	9765	1970	687	1889977	262072	99,19	95,78
inst67		1	1963	–	44	134169	34090	1	1963	1963	48	44	143273	36884	1963	3236	3126766	254955	98,67	98,52
inst68		0	1975	–	30	64077	10523	0	1975	1975	30	30	64383	10635	1894	–	7883317	955270	99,17	99,17
inst69		0	1974	–	17	52308	11045	0	1974	1974	17	17	52308	11045	1974	291	689032	100403	99,53	94,16
inst70		0	1970	–	6	18333	5276	0	1970	1970	6	6	19028	5513	1970	377	938618	128983	99,83	98,41
inst71	25	1	2449	–	47	163953	28159	1	2449	2449	47	47	164066	28211	2449	392	896901	96825	98,69	88,01
inst72		1	2453	–	38	57709	12630	1	2453	2453	39	38	58647	12922	2453	1101	2780985	326216	98,92	96,46
inst73		0	2450	–	7	15962	3783	0	2450	2450	7	7	16059	3828	2450	141	501142	114711	99,81	95,04
inst74		1	2469	–	29	89184	18948	0	2469	2468	33	24	99460	20910	2469	985	2253889	332873	99,08	96,65
inst75		2	2449	–	134	91445	19956	2	2449	2449	140	133	96464	21507	1681	–	2776043	152673	96,11	96,11
inst76		1	2450	–	78	213959	40375	1	2450	2450	79	78	216154	41312	2450	912	3049901	692151	97,81	91,34
inst77		1	2447	–	18	58036	15557	1	2447	2447	18	18	58085	15581	2447	438	1364534	284601	99,50	95,89
inst78		0	2445	–	130	259134	37187	0	2445	2445	131	129	263275	38837	2445	2029	4956591	555685	96,36	93,54
inst79		2	2458	–	6	6794	2530	2	2458	2458	12	6	8592	3145	2154	–	9713526	993043	99,67	99,67
inst80		2	2446	–	24	48996	13623	2	2446	2446	28	24	49826	13942	1892	–	6331371	634550	99,22	99,22

TABLE 5.3: The runtime in second for SMSDP 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.

		Basic *S&D*				Improved *S&D*						*B&B*				GAP	
	|y|	L_p	Obj	CPU	FindObj	Backtracks	Nodes	L_p	Obj	Obj_{L_p}	CPU	FindObj	Backtracks	Nodes	Obj	CPU	Backtracks	Nodes	*S&D/BS&D*	*S&D/B&B*
0	10	0	209	54	10	60560	118371	0	209	209	10	10	60560	118371	209	572	2919885	9750028	81,48	98,25
1		0	209	158	116	534649	1121114	0	209	209	116	116	534649	1121114	209	498	3213152	8090107	26,58	76,71
2		1	187	53	11	60207	116601	0	187	189	37	8	204625	448493	187	251	1567333	3906182	30,19	85,26
3		0	201	63	19	98809	199586	0	201	201	19	19	98809	199586	201	102	642584	1619182	69,84	81,37
4		0	221	163	120	585378	1211618	0	221	221	120	120	585378	1211618	221	288	1877107	4354403	26,38	58,33
5	11	1	231	157	65	358814	781328	0	231	234	70	26	360294	747686	231	1101	5995262	15767807	55,41	93,64
6		1	197	123	32	197715	422444	0	197	199	86	18	452029	1035082	197	217	1143785	3052372	30,08	60,37
7		1	222	113	26	144750	292322	0	222	223	50	19	263089	552315	222	2239	11850671	35428012	55,75	97,77
8		0	183	98	8	61542	113275	0	183	183	8	8	61542	113275	183	83	420474	1367753	91,84	90,36
9		1	236	123	27	152659	301098	0	236	239	102	13	527335	1190066	236	302	1698370	3746299	17,07	66,23
10	12	1	209	266	59	362562	727315	0	209	213	76	39	420511	835869	209	1020	5099365	14281235	71,43	92,55
11		1	254	1013	814	4039001	8488363	0	254	255	1446	213	6727790	14681260	257	–	18743657	48622555	-42,74	59,83
12		0	207	255	56	314329	687610	0	207	207	56	56	314329	687610	207	1371	6806487	20784515	78,04	95,92
13		0	221	267	67	161778	314242	0	221	221	67	67	161778	314242	221	227	1223429	2706915	74,91	70,48
14		0	209	248	39	108970	225269	0	209	209	39	39	108970	225269	209	292	1549413	3842850	84,27	86,64
15	13	0	221	574	141	518906	1046603	0	221	221	141	141	518906	1046603	221	681	3414545	7434312	75,44	79,30
16		0	241	540	86	467618	926741	0	241	241	86	86	467618	926741	241	340	1720092	3708438	84,07	74,71
17		0	235	465	15	49116	95328	0	235	235	15	15	49116	95328	235	169	841864	1904622	96,77	91,12
18		0	177	481	33	71770	127547	0	177	177	33	33	71770	127547	177	154	758339	1730981	93,14	78,57
19		0	213	525	89	395196	821175	0	213	213	89	89	395196	821175	213	401	1774810	6368353	83,05	77,81
20	14	0	188	210	16	68875	125028	0	188	188	16	16	68875	125028	188	320	1576305	5020386	92,38	95,00
21		0	228	1117	142	671076	1326115	0	228	228	142	142	671076	1326115	228	488	2290737	5143937	87,29	70,90
22		0	226	1139	159	603011	1213031	0	226	226	159	159	603011	1213031	226	577	2572092	5485536	86,04	72,44
23		0	248	1027	44	178770	352593	0	248	248	44	44	178770	352593	248	162	745537	1569241	95,72	72,84
24		0	290	1077	194	481248	978557	0	290	290	194	194	481248	978557	290	877	3861844	7947852	81,99	77,88
25	15	0	214	2212	91	176688	349372	0	214	214	91	91	176688	349372	214	268	1169161	2338777	95,89	66,04
26		0	255	2218	92	266762	529558	0	255	255	92	92	266762	529558	255	161	689171	1373246	95,85	42,86
27		0	228	2110	73	96926	189806	0	228	228	73	73	96926	189806	228	2624	11053562	26493166	96,54	97,22
28		0	238	2126	134	511548	1044352	0	238	238	134	134	511548	1044352	238	361	1470306	3050300	93,70	62,88
29		0	185	2160	71	320735	632667	0	185	185	71	71	320735	632667	185	259	1194437	2381561	96,71	72,59

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.

First class						Second class					
depth	1	2	3	4	5	depth	1	2	3	4	5
ins1	8	10	35	62	62	ins41	33	10	20	98	126
ins2	3	4	42	87	87	ins42	3	3	5	26	52
ins3	41	45	73	101	126	ins43	2	2	27	75	102
ins4	3	12	37	37	37	ins44	3	3	19	71	71
ins5	5	7	32	58	58	ins45	4	4	14	41	68
ins6	2	1	23	49	49	ins46	0	9	13	40	67
ins7	7	12	38	38	38	ins47	4	14	21	30	54
ins8	8	7	33	34	34	ins48	6	2	12	39	39
ins9	2	15	66	118	118	ins49	7	2	10	33	61
ins10	0	1	26	26	26	ins50	8	7	15	41	97
ins11	132	216	–	–	–	ins51	54	9	629	–	–
ins12	3	11	664	–	–	ins52	79	433	446	–	–
ins13	22	266	996	–	–	ins53	3	134	555	–	–
ins14	3	54	813	–	–	ins54	7	9	139	–	–
ins15	7	2	425	–	–	ins55	24	10	17	818	–
ins16	8	7	447	–	–	ins56	70	140	305	–	–
ins17	6	258	–	–	–	ins57	16	49	47	937	–
ins18	12	8	454	–	–	ins58	33	90	126	408	–
ins19	20	136	763	–	–	ins59	15	3	137	418	–
ins20	2	5	899	900	901	ins60	16	20	60	735	–
ins21	55	157	–	–	–	ins61	33	110	20	–	–
ins22	78	43	–	–	–	ins62	119	126	–	–	–
ins23	19	8	–	–	–	ins63	18	254	208	–	–
ins24	89	595	–	–	–	ins64	58	174	–	–	–
ins25	63	116	–	–	–	ins65	12	5	–	–	–
ins26	61	78	–	–	–	ins66	29	181	–	–	–
ins27	50	–	–	–	–	ins67	47	121	25	–	–
ins28	9	170	–	–	–	ins68	31	9	–	–	–
ins29	3	161	–	–	–	ins69	17	3	620	–	–
ins30	5	–	–	–	–	ins70	6	3	629	–	–
ins31	55	–	–	–	–	ins71	47	45	–	–	–
ins32	110	92	–	–	–	ins72	39	312	–	–	–
ins33	30	–	–	–	–	ins73	7	452	–	–	–
ins34	133	–	–	–	–	ins74	33	–	–	–	–
ins35	14	442	–	–	–	ins75	141	146	–	–	–
ins36	87	–	–	–	–	ins76	78	99	–	–	–
ins37	7	92	–	–	–	ins77	18	16	–	–	–
ins38	107	–	–	–	–	ins78	132	482	917	–	–
ins39	52	5	–	–	–	ins79	12	40	–	–	–
ins40	7	–	–	–	–	ins80	27	66	–	–	–

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.

depth	1	2	3	4	5
ins1	10	8	15	67	68
ins2	116	89	96	102	152
ins3	37	42	42	40	92
ins4	19	15	26	31	37
ins5	120	204	110	141	271
ins6	70	66	503	666	664
ins7	86	92	111	105	227
ins8	50	69	82	87	193
ins9	8	4	2	9	14
ins10	102	77	184	187	187
ins11	76	301	308	308	308
ins12	–	–	–	–	–
ins13	56	142	383	383	383
ins14	67	95	142	132	109
ins15	39	64	134	150	163
ins16	141	226	117	173	151
ins17	86	83	123	97	116
ins18	15	82	27	528	551
ins19	33	30	26	41	56
ins20	89	72	71	574	574
ins21	16	19	21	55	321
ins22	142	112	193	107	154
ins23	159	196	188	224	294
ins24	44	58	419	223	–
ins25	194	236	334	416	–
ins26	91	147	–	–	–
ins27	92	272	–	–	–
ins28	73	431	288	–	–
ins29	134	146	372	861	601
ins30	71	41	49	102	73

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 *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 *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 $S\&D$. Several lines can be distinguished:

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

Bibliography

Pascal Van Hentenryck. *The OPL Optimization Programming Language*. MIT Press, Cambridge, MA, USA, 1999. ISBN 0-262-72030-2.

Jean-Guillaume Fages, Narendra Jussien, Xavier Lorca, and Charles Prudhomme. Choco3: an open source java constraint programming library. Technical report, Research report 13/1/INFO, Ecole des Mines de Nantes, 2013. to appear, 2013.

Nicholas Nethercote, Peter J Stuckey, Ralph Becket, Sebastian Brand, Gregory J Duck, and Guido Tack. Minizinc: Towards a standard cp modelling language. In *Principles and Practice of Constraint Programming–CP 2007*, pages 529–543. Springer, 2007.

Alan M Frisch, Warwick Harvey, Chris Jefferson, Bernadette Martínez-Hernández, and Ian Miguel. Essence: A constraint language for specifying combinatorial problems. *Constraints*, 13(3):268–306, 2008.

F. Rossi, F. Van Beek, and T. Walsh. *Handbook of Constraint Programming*. Elsevier Science Ltd, Foundations of Artificial Intelligence, Radarweg 29, PO Box 211, 1000 AE Amsterdam, The Netherlands, 2006a.

Krzysztof Apt. *Principles of Constraint Programming*. Cambridge University Press, New York, NY, USA, 2003. ISBN 0521825830.

Edward Tsang. *Foundations of Constraint Satisfaction*. Academic Press Limited, London and San Diego, 1993. ISBN 0–12–701610–4.

Christian Schulte and Peter J. Stuckey. Efficient constraint propagation engines. *ACM Trans. Program. Lang. Syst.*, 31(1):2:1–2:43, 2008.

Francesca Rossi, Peter van Beek, and Toby Walsh. *Handbook of Constraint Programming (Foundations of Artificial Intelligence)*. Elsevier Science Inc., New York, NY, USA, 2006b. ISBN 0444527265.

Willem-Jan van Hoeve and Irit Katriel. *Handbook of Constraint Programming*, chapter Global Constraints. Elsevier, 2006.

Jean-Charles Rgin. Global constraints and filtering algorithms, 2003.

Jean-Charles Rgin. *Hybrid Optimization: the 10 years of CP-AI-OR*, chapter Global Constraints: a Survey, page in press. Springer, 2010.

Torsten Fahle and Meinolf Sellmann. Cost based filtering for the constrained knapsack problem. *Annals OR*, 115(1-4):73–93, 2002.

P. Schaus. *Solving Balancing and Bin-Packing problems with Constraint Programming*. PhD thesis, University of Maryland. Technical Report for the Institute for Systems Research, 2009.

Michael A. Trick. A dynamic programming approach for consistency and propagation for knapsack constraints. *Annals OR*, 118(1-4):73–84, 2003a.

D.S. Chen, R.G. Batson, and Y. Dang. *Applied Integer Programming: Modeling and Solution*. Wiley, 2011.

H. Kellerer, U. Pferschy, and D. Pisinger. *Knapsack Problems*. Springer, 2004. ISBN 9783540402862.

S. Martello and P. Toth. *Knapsack problems: Algorithms and computer implementations*. DEIS, University of Bologna, John Wiley & Sons Ltd, Baffins Lane, Chichester West Sussex - PO19 1UD, England, 1990.

A. Freville. The multidimensional 0-1 knapsack problem: An overview. *European Journal of Operational Research*, 155(1):1–21, 2004.

P. Cappanera and M. Trubian. A local-search-based heuristic for the demand-constrained multidimensional knapsack problem. *INFORMS Journal on Computing*, 17(1):82–98, 2005.

D. Pisinger. A minimal algorithm for the multiple-choice knapsack problem. *European Journal of Operational Research*, 83(2):394–410, 1995.

M. Moser, D.P. Jokanovic, and N. Shiratori. An algorithm for the multidimensional multiple-choice knapsack problem. *IEICE Transactions on Fundamentals of Electronics*, 80:582–589, 1997a.

S. Khan. *Quality adaptation in multi-session adaptative multimedia system: Model and architecture*. PhD thesis, Departement of Electronical and computer Engineering, University of Victoria, 1998.

V.C. Li. Tight oscillation tabu search for multidimnesional knapsack problems with generalized upper bound constraints. *Computer and Operations Research*, 32:2843–2852, 2005.

V.C. Li and G.L. Curry. Solving multidimensional knapsack problems with generalized upper bound constraints using critical event tabu search. *Computer and Operations Research*, 32:825–848, 2005.

J. Beasley. Or-library: distributing test problems by electronic mail. *Journal of the Operational Research Society*, 41:1069–1072, 1990.

P.C. Chu and J.E. Beasley. A genetic algorithm for the multidimensional knapsack problem. *Journal of Heuristics*, 4:63–86, 1998.

S. Khan, F. Kin, E. Manning, and M. Akabr. Solving the knapsack problem for adaptative multimedia system. *Studia Informatica - Special Issue on Combinatorial Problems*, 2:154–174, 2002.

B. Han, J. Leblet, and G. Simon. Hard multidimensional multiple choice knapsack problems, an empirical study. *Computer and Operations Research*, 37:172–181, 2010a.

Christian Bessière, Emmanuel Hebrard, Brahim Hnich, Zeynep Kiziltan, and Toby Walsh. The roots constraint. In *CP*, pages 75–90, 2006.

Jean-Charles Régin. Arc consistency for global cardinality constraints with costs. In *CP*, pages 390–404, 1999.

Jean-Charles Régin. A filtering algorithm for constraints of difference in csps. In *AAAI*, pages 362–367, 1994.

Gilles Pesant. A regular language membership constraint for finite sequences of variables. In *CP*, pages 482–495, 2004.

Michael A. Trick. A dynamic programming approach for consistency and propagation for knapsack constraints. *Annals OR*, 118(1-4):73–84, 2003b.

Meinolf Sellmann. Approximated consistency for knapsack constraints. In *CP*, pages 679–693, 2003.

Meinolf Sellmann. The practice of approximated consistency for knapsack constraints. In *AAAI*, pages 179–184, 2004.

Vipul Jain and Ignacio E Grossmann. Algorithms for hybrid milp/cp models for a class of optimization problems. *INFORMS Journal on computing*, 13(4):258–276, 2001.

E. L. Lawler and D. E. Wood. Branch-And-Bound Methods: A Survey. *Operations Research*, 14(4):699–719, 1966. ISSN 0030364X. doi: 10.2307/168733.

Martin Moser, Dusan P. Jokanovic, and Norio Shiratori. An algorithm for the multidimensional multiple-choice knapsack problem. *IEICE TRANSACTIONS on Fundamentals of Electronics, Communications and Computer Sciences*, 80 (3):582–589, 1997b.

Alan M. Frisch, Ian Miguel, and Toby Walsh. Modelling a steel mill slab design problem. In *Proceedings of the IJCAI-01 Workshop on Modelling and Solving Problems with Constraints*, pages 39–45, 2001.

Bing Han, Jimmy Leblet, and Gwendal Simon. Hard multidimensional multiple choice knapsack problems, an empirical study. *Computers & operations research*, 37(1):172–181, january 2010b.

Willem Jan van Hoeve and Michela Milano. Decomposition based search - a theoretical and experimental evaluation. *CoRR*, cs.AI/0407040, 2004.

Michela Milano and Willem J. van Hoeve. Reduced cost-based ranking for generating promising subproblems. In *Proceedings of the 8th International Conference on Principles and Practice of Constraint Programming*, CP '02, pages 1–16, London, UK, 2002. Springer-Verlag. ISBN 3-540-44120-4. URL http://portal.acm.org/citation.cfm?id=647489.727164.

William D. Harvey and Matthew L. Ginsberg. Limited discrepancy search. In *Proceedings of the 14th international joint conference on Artificial intelligence - Volume 1*, pages 607–613, San Francisco, CA, USA, 1995. Morgan Kaufmann Publishers Inc.

Matthew Kitching and Fahiem Bacchus. Exploiting decomposition on constraint problems with high tree-width. In *IJCAI 2009, Proceedings of the 21st International Joint Conference on Artificial Intelligence, Pasadena, California, USA, July 11-17, 2009*, pages 525–531, 2009.

Jean-Charles Régin, Mohamed Rezgui, and Arnaud Malapert. Improvement of the embarrassingly parallel search for data centers. In *Principles and Practice of Constraint Programming - 20th International Conference, CP 2014, Lyon, France, September 8-12, 2014. Proceedings*, pages 622–635, 2014.