REPUBLIQUE ALGERIENNE DEMOCRATIQUE ET POPULAIRE
DEMOCRATIC AND POPULAR REPUBLIC OF ALGERIA
MINISTRY OF HIGHER EDUCATION AND SCIENTIFIC RESEARCH

UNIVERSITY OF MOHAMED BOUDIAF - MSILA
FACULTY OF MATHEMATICS AND COMPUTER SCIENCES

DEPARTEMENT OF MATHEMATICS

MASTER MEMORY

Faculty: Mathematics and Computer science
Department: Mathematics
Specialty: Discrete Mathematics

Submitted by :
Taki Eddine *BACHIRI*

TITLE

The knapsack problem: Definition and Resolution

Publicly supported: 04/06/2016. **The jury is composed as follows:**

Mr. Lamiche Chaabane	**MCA. University of Msila**	**President**
Mr. Belouadah Hocine	**Prof. University of Msila**	**Supervisor**
Mr.Yakoubi Rachad	**MAA. University of Msila**	**Examiner**
Dr. Bounab Noura	**MCB. University of Msila**	**Examiner**

Promotion : 2015 /2016

Acknowledgements

All Praise is to Allah, The most Graceful and most Compassionate the Almighty, who gave me strength and good health for accomplishing this work.

* * * * * *

My sincere gratitude is addressed to PhD. Belouadah - faculty of the Mathematics- at University of M'sila the supervisor of this dissertation, for his patience, his endless advices, his brilliant ideas and his genuine efforts. His door was all time open and he never failed in providing me with academic advice and moral support during the challenging time of preparing this study.

I would also like to extend my thankfulness to all students of my classmates as the second readers of this and I am gratefully indebted to their valuable comments on this thesis.

Finally, I must express my very profound gratefulness to my dear parents and to my friend Lyazid for providing me with unfailing support and continuous encouragement throughout my years of study and through the process of researching and writing this dissertation. This accomplishment would not have been possible without them.

Abstract

The knapsack problem is a COP problem (Combinatorial Optimization Problem) derived from artificial intelligence fields. This problem can be seen as NP (Non Probabilistic Problem) because there is an exponential combination of instances, where the entirety set can't be solved by any technique or method.
There are variations relating to this problem: problem with limited number of products, problems with unlimited number of products, multiple choice problems, choosing a different product from some categories, weight's products problem, economical amount problem, among others. This dissertation aims at giving an overview of this problem and its application in real life.

Resumé

Le problème de sac à dos est un problème d'optimisation combinatoire dérivé de champs d'intelligence artificielle. Ce problème peut être considéré comme NP, car il est une combinaison exponentielle des cas, lorsque l'ensemble de totalité ne peut pas être résolu par toute technique ou méthode.
Il y a des variations relatives à ce problème: problème avec nombre limité de produits, des problèmes avec un nombre illimité de produits, problèmes de choix multiples, le choix d'un produit différent de certaines catégories, les produits du problème de poids, problème de quantité économique, entre autres. Cette thèse vise à donner un aperçu de ce problème et son application dans la vie réelle.

Contents

List of Figures

List of Tables

List of Algorithms

Introduction

A famous class of combinatorial optimization problems is known as the knapsack problem. Specifically, this class of combinatorial optimization problems characterizes a class of integer linear programming and is then classified as NP–hard problems due to their complexity degree.

The basic aim of knapsack problem is to find a combination of different objects that a thief chooses for his knapsack such that total value of all objects of his/her choice is maximized. This situation is common in complexity theory, cryptography, business, combinatorics, and applied mathematics. Owing to its vast applications, knapsack problem has been rigorously studied by both theorists and experimenters.

The problem has equally been modeled into many industrial situations, such as cargo loading and capital budgeting, logistics like loading of airplanes or boats, the economy such as financial management. The class of knapsack problems is very wide and includes many sub-problems, and, there is no text in the literature of combinatorial optimization that covers and fully treats it. However, there is a great number of books and researches that threw light on several classic problems of the family, treat some more specific generalizations of the problems, or give a profound introduction (see e.g., Ibraki 1987, Pisinger 1995, Dudzinski and Walukiewicz 1987, Martello and Toth, 1990 and Syslo, 1983).

The specific problem that arises depends on the number of knapsacks (single or multiple) to be filled and on the number of available items of each type (bounded or unbounded). Because of their wide range of applicability, knapsack problems have known a large number of variations such as: single and multiple-constrained knapsacks, knapsacks with disjunctive constraints, multidimensional knapsacks, multiple choice knapsacks, single and multiple objective knapsacks, integer, linear, non-linear knapsacks, deterministic and stochastic knapsacks, knapsacks with convex or concave objective functions, etc.

The literature in the knapsack problems has been dominated by the analysis of problem with binary variables, the 0–1 knapsack problem, since the pioneering work of Dantzing in the late 50's (Dantzing, 1957). Since then, a number of different approaches for solving the knapsack problems have been proposed. The 0–1 knapsack problem has attracted special interest. As knapsack problems (in particular 0-1 knapsack problem) are classified as NP–hard problems due to their complexity degree, there is no exact methods for solving the problem other than the enumeration space approaches. However, a wide variety of inexact approaches, including branch and bound, dynamic programming, have been proposed in the literature of integer programming for solving knapsack problems (see Ibraki, 1987).

The objective

The principal objective of this dissertation is to introduce a general definition for the knapsack problem and through applying some exact or approximate methods for solving this problem.

Organization of the dissertation

as regards the planning of the study, this dissertation is organized into three main chapters.

- In chapter 1, we give a general overview of combinatorial optimization problem, and some notions of complexity of algorithms theory, and some methods for solving this problem. Then, we give the mathematical representation of the knapsack problems and review the related literature.

- In chapter 2, we are interested particularly in classical resolution methods, such as dynamic programming and Branch and Bound and approaches as well as methods, we present greedy algorithm, simulated annealing, and fully polynomial time approximation scheme.

- Finally, we present in Chapter 3 analyzes and results of applying the resolution methods for the knapsack problem. We finish this memoir by presenting our general conclusions and research perspectives.

Chapter 1

The knapsack problem

1.1 Introduction

In the field of mathematics and computer science, one of the most challenging problems is searching for optimal arrangements of elements within hundreds of thousands of possibilities. Collectively, these problems are known as combinatorial optimization problems (COPs), and their importance lies in the large diversity of real-life problems that can be solved in the industry and engineering sectors by studying them.
They become a challenge because using a direct approach of calculating all the possibilities and choosing the best one becomes an unfeasible task, because the number of possible combinations grows quickly compared to checking modestly sized instances, which is beyond the capabilities of even the most powerful today's super computers, therefore a great deal of research has been invested in developing faster and better algorithms for solving COPs.

There are many combinatorial optimization problems, one of these problems is knapsack problem which is the subject of my dissertation. The knapsack problem is a classic and widely studied problem in combinatorial optimization arising from resource allocation[**1**] . In its original form there are multiple items whose weights and rewards are known in advance, and we want to choose a subset of these items to maximize the sum of the rewards of the selected items subject to the knapsack capacity constraint for the items' total weights. Knapsack problems naturally arise in resource allocation and budget planning problems where one wants to extract the maximum return from candidate projects while keeping the resource/budget balanced.

In this chapter, we briefly introduce computational optimization, some notions of complexity theory, some methods of resolution of these problems,then we define the knapsack problem and some problems belonging to the family of the knapsack problem.

1.2 Combinatorial optimization problem

Combinatorial optimization means searching for an optimal solution in a finite or countably infinite set of potential solutions. Optimality is defined with respect to some criterion function, which is to be minimized or maximized.

Definition 1.2.1. *A **Combinatorial optimization problem** can be defined by :*

- *Variable vector* $x = (x_1, x_2, ..., x_n)$,
- *Domain of variables* $D = (D_1, D_2, ..., D_n)$ *, where the* $(D_i)_{i=1...,n}$ *are finite sets,*
- *Set of constraints* Ω,
- *An objective function f to minimize or maximize,*
- *Set of all possible solutions is feasible* $S = \{x = (x_1, x_2, ..., x_n), \in D/$ *x satisfies all the constraints* $\}$ *, the set S is also called a search space.*
- *a problem*

$$(COP) : \begin{cases} \textbf{opt} \quad f(x) \\ x \in S \end{cases}$$

*where '**opt**' is replaced by either '**min**' or '**max**'.*

Usually, such problems can be formulated as Integer Programs with binary variables, which indicate for each member of the collection, whether it belongs to the subset or not.

A lot of problems fit into this definition. For example partitioning, assignment, covering, scheduling, shortest path, travelling salesman, spanning tree, matching, etc.

Definition 1.2.2. ***Instance*** **[2]**
An instance I of a minimization problem is a pair (X, f) where $X \subseteq S$ *is a finite set of feasible solutions, and a cost function f (or objective) to minimize* $f : X \longrightarrow R$ *. The problem is to find* $s^* \in X$ *telque* $f(s^*) \leq f(s)$ *for any element* $s \in X$.

Definition 1.2.3. ***Constraint***
A constraint of a problem is a restriction imposed by the nature and characteristics of the problem on the proposed solutions.

Definition 1.2.4. ***The search space***
The search space S of a problem consists of the set of values that can be taken by the variables $(s_1, s_2, ..., s_n)$ *that construct s solution.*

Definition 1.2.5. ***The neighborhood*** *V(s) of solution s is a subset of S, whose members are solutions close (adjacent) of the solution s. Indeed, we say that solution* $<s'>$ *is a neighbor of s, if it can be obtained by slightly modifying the s solution.*

Remark 1.2.1. *Indeed a maximization problem can be easily turned into a problem minimization considering the following equivalence : Maximize* $f(x) \Leftrightarrow$ *Minimize* $-f(x)$

Solving a combinatorial optimization problem is usually to find the best within the meaning of criterion f, feasible solution. This is what is called an optimal solution.

Definition 1.2.6. ***Optimal Solution***
Optimal Solution s^* *is a feasible solution such that there is no other feasible solution* s *of* S *to lower cost (or minimizing) than* s^*.

- **global minimum** : a point s^* is called global minimum of the function f if : $\forall s, s^* \Rightarrow f(s^*) < f(s)$
- **strong local minimum** : a point s^* is called strong local minimum of the function f if : $\forall s \in v(s^*), s^* \Rightarrow f(s^*) < f(s)$ où $v(s^*)$ represents the neighborhood of s^*
- **weak local minimum** : a point s^* is called weak local minimum of the function f if : $\forall s \in v(s^*), s^* \Rightarrow f(s^*) \leq f(s)$

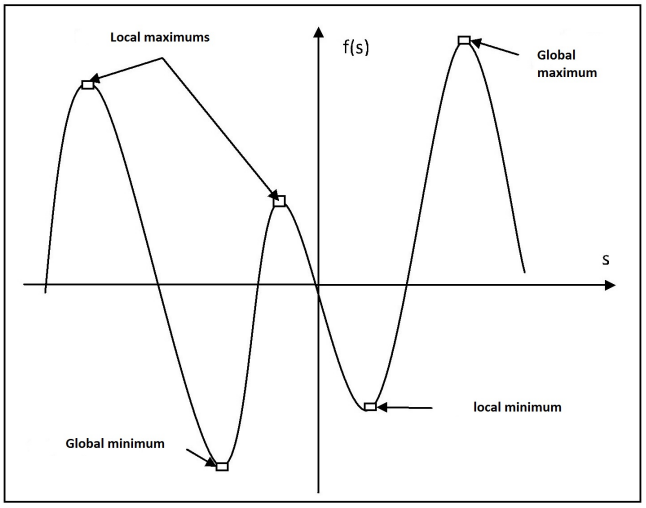

FIGURE 1.1 – Curve representing the local optima and global optima

1.3 Basic notions of complexity theory

A computational problem is a mathematical problem, defined by some parameters and one or more questions, that a computer has to solve, i.e., for which we want to find a solution. For solving efficiently a problem, an algorithm has to be designed, understood as a finite procedure expressed in terms of predefined elementary operations.

The efficiency of an algorithm is given by the amount of resources required to execute it, as time and memory. The most crucial efficiency measure is the running time needed by an algorithm for finding a solution for any instance of a problem, that is given on the input of the algorithm. This time depends on the size of the input, denoted as I . If the input size is $size(I) = n$, it means that n elements are required to describe the problem parameters. The size can be expressed in bytes or bits, but in general it is given informally.

The time required by an algorithm to solve any instance of a problem of size n is a function of n, which is called the algorithm complexity. The big-O notation is commonly used to describe this function. An algorithm for which this function can be bounded from above by a polynomial in n $(O(n), O(nlogn), O(n^2), etc.)$ is called a polynomial time algorithm. An algorithm for which this function cannot be bounded by such a polynomial function $(2^n, n!, n^n, etc.)$ is called an exponential time algorithm.

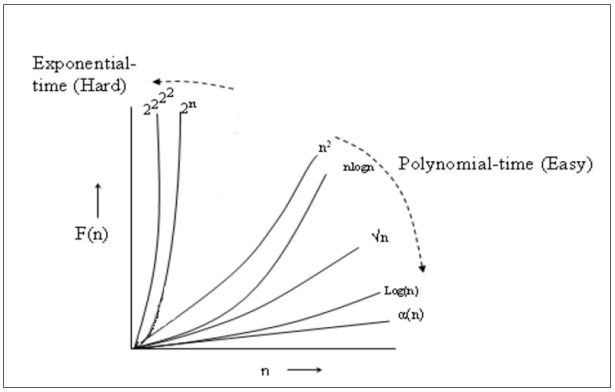

FIGURE 1.2 – The graphical representation of NP and P

Definition 1.3.1. **[3]** *Let f and g be functions from positive integers to positive integers.*

- *We say f is $O(g(n))$ (read: 'or f is order g') if g is an upper bound on f: there exists a fixed constant c and a fixed n_0 such that for all $n \geq n_0$,*
  we have $f(n) \leq c.g(n)$
  Equivalently, f is O(g(n)) if the function f(n)/g(n) is bounded above by some constant.see(a)

- *We say that f is $\Omega(g(n))$ (read: 'f is omega of g') if g is a lower bound on f for large n. Formally, f is $\Omega(g)$ if there is a fixed constant c and a fixed n_0 such that for all $n \geq n_0$, we have $c.g(n) \leq f(n)$. see(b)*

- *We say that f is $\Theta(g(n))$ (read: 'f is theta of g') if g is an accurate characterization of f for large n: it can be scaled so it is both an upper and a lower bound of f. That is, f is both $O(g(n))$ and $\Theta(g(n))$). Expanding out the definitions of Ω and O, f is $\Theta(g(n))$ if there are fixed constants c_1 and c_2 and a fixed n_0 such that for all $n \geq n_0$, we have $c_1.g(n) \leq f(n) \leq c_2.g(n)$. see(c)*

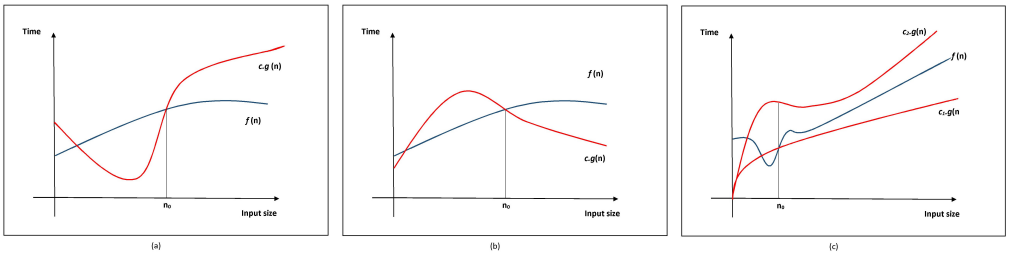

FIGURE 1.3 – The graphical representation

A problem which is so hard that no polynomial time algorithm can possibly solve it is called intractable. Algorithms that have running times polynomial in n and the maximum of the elements of the instance, denoted $\max(I)$, are called pseudopolynomial. The complexity class of a problem indicates its intrinsic difficulty, and this information is crucial for being able to solve the problem properly. This is the object of the theory of NP-completeness, presented in the seminal book of Garey and Johnson (1979)[2].

The Theory of NP-completeness applies on several types of problems. We distinguish the following two main classes of problems:

* **Decision problems** that are defined by a name, an instance, which is a description of all the parameters, and a question for which the answer belongs to the set $\{yes, no\}$.
* **Optimization problems** that are defined by a name, an instance, and in which the aim is to find a solution with minimum value of a given function.

There exist two main classes of decision problems.

Definition 1.3.2. ***The class P*** *is the class of decision problems that can be solved by a polynomial time algorithm running on a deterministic computer in which at any time there may be done at most one action, contrary to non-deterministic computer in which more actions may be done. These problems are said to be 'easy' to solve, in the sense that large instances can possibly be solved in a reasonable computation time.*

Definition 1.3.3. ***The class NP*** *is the class of decision problems that can be solved by a polynomial time algorithm running on a nondeterministic computer. It is equivalent to say that the class NP is the class of decision problems for which a response 'yes' can be verified in polynomial time on a deterministic computer. It is clear that $P \subset NP$.*

Definition 1.3.4. *A polynomial time algorithm is an algorithm whose time complexity is $O(p(n))$, where p is a polynomial function and n is the size of proceedings (or the input length) . If k is the greatest exponent of this polynomial in n, the corresponding problem is said to be solvable in $O(n^k)$ and belongs to the class P , an example of a problem polynomial is one of the connection in a graph.*

Definition 1.3.5. *We say that a decision problem P_1 polynomially reduces to a decision problem P_2 if and only if there exists a polynomial time algorithm f , which can build, from any instance I_1 of P_1, an instance $I_2 = f(I_1)$ of P_2 such that the response to problem P1 for instance I1 is 'yes' if and only if the answer to problem P_2 for instance I_2 is 'yes'.*

Definition 1.3.6. ***NP-Completeness*** *A problem P is NP-complete if P belongs to NP and any problem of N P polynomially reduces to P .*

*The NP-complete problems create a subset of NP and we say that NP-complete problems are the 'difficult' problems of NP. Indeed, let us suppose that an NP-complete problem can be solved in polynomial time. Then, according to (**Definition 1.3.5**), any problem in NP can be solved in polynomial time, which implies that $NP \subset P$ and thus $P = NP$. Therefore, unless $P = NP$, the NP-complete problems cannot be solved in polynomial time, which make these problems more 'difficult' that those from the class P.*

Definition 1.3.7. ***NP-Hardness***. *Intuitively, these are the problems that are at least as hard as the NP-complete problems. Note that NP-hard problems do not have to be in NP, and they do not have to be decision problems. The precise definition here is that a problem X is NP-hard, if there is an NP-complete problem Y, such that Y is reducible to X in polynomial time*[**22**]

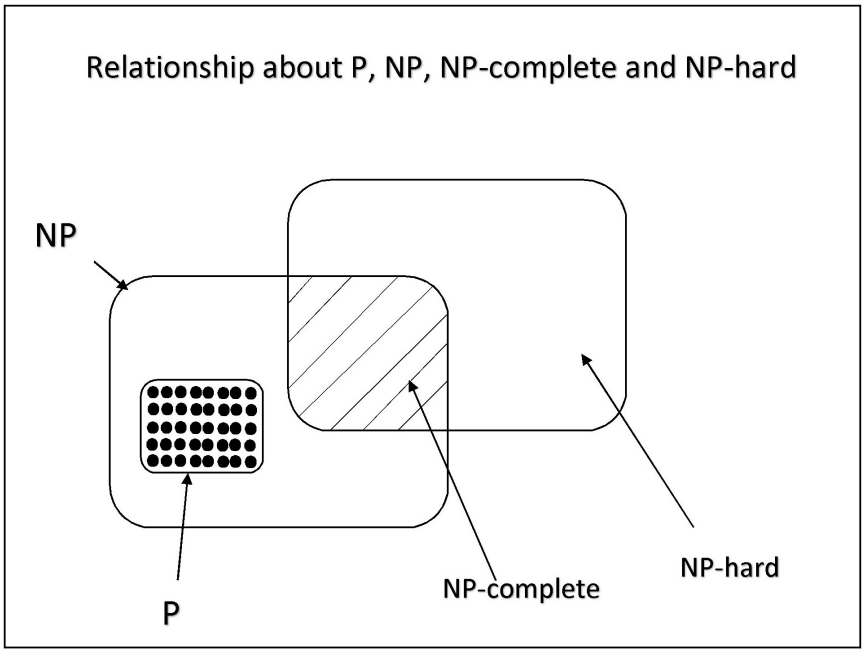

FIGURE 1.4 – Relation about P, NP, NP-complete,NP-hard

1.4 Some methods of resolution

For solving an optimization problem,there are many kinds of approach. An exact method is an algorithm which allows the obtaining of at least an optimal solution of the problem to solve. Once an optimal obtained solution,the algorithm also has to be capable of proving the optimality, what can sometimes be also difficult to make that to obtain this solution. This kind of algorithm generally requires an important execution time and/or important resources memories on authorities of big size. The complexity of these algorithms is thus often too important so that they can be applied to problems including of many variables and/or constraints. Without getting into detail, we can quote some classic exact approaches used in operational research on which we return during this report:

- Branch and bound
- Dynamic Programming

When the used algorithm does not inevitably allow to obtain an optimal solution of the problem, the method is called approximate or heuristic method. This kind of approach is

often adapted to the problem to be solved, and it generally allows the generation of a realizable solution of good quality into cubes not very important computing times and also requiring a load limited memory.
There are other methods that are placed on a more general level, and involved in all situations where the engineer is no known effective heuristic to solve a given problem, these methods are metaheuristics. In 2006, the network Metaheuristic (*metaheuristics.org*) defines metaheuristics as a set of concepts used to define heuristic methods, can be applied to a wide variety of problems.

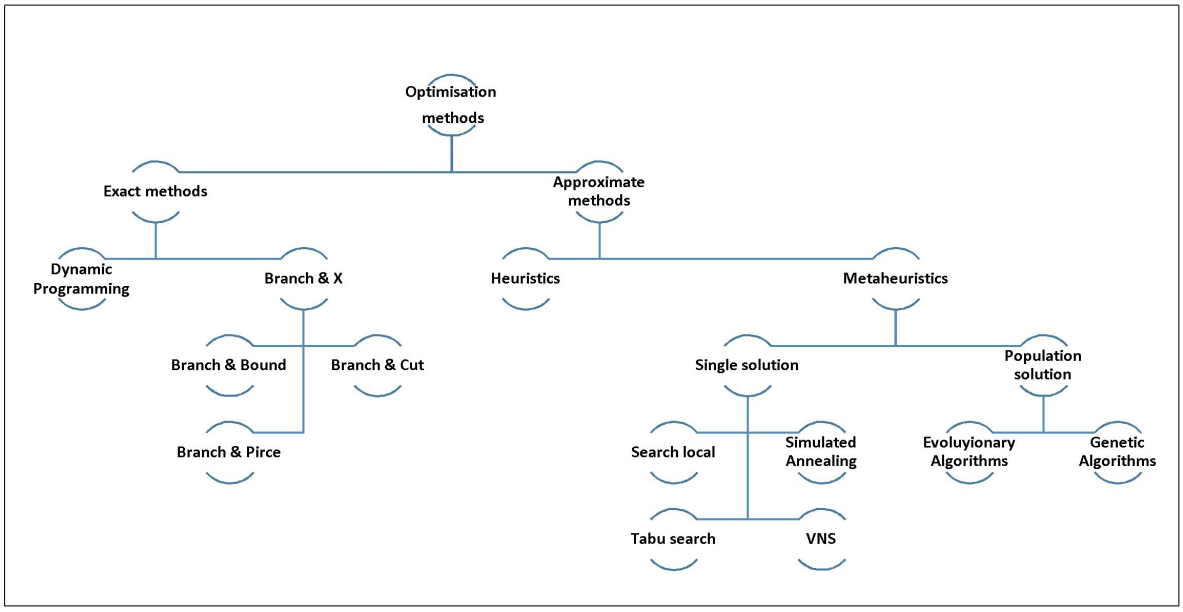

FIGURE 1.5 – Classification of resolution methods

1.4.1 Exact methods

In this sub-section we describe very briefly two general methods of solving many combinatorial problems , namely the method of dynamic programming and the method of branch and bound. The essential principle of generally exact method is to enumerate, often implicitly, all the combinations of the search space. Among the exact methods we find separation and evaluation, said branch and bound methods, and dynamic programming.

Dynamic Programming

Fundamentals of dynamic programming were elaborated by Bellman in the 1950's and presented in [**5**]; [**6**] . The name "Dynamic Programming" is slightly misleading, but generally accepted. A better description would be "recursive" or "multistage" optimization, since it interprets optimization problems as *multistage decision processes*. It means that the problem is divided into a number of stages, and at each stage a decision is required which impacts on the decisions to be made in later stages, and at each stage a decision is required which impacts on the decisions to be made in later stages. Now, Bellman's principle of optimality is applied to draw up a recursive equation which describes the optimal criterion value at a

given stage in terms of the previously obtained one.
This principle can be formulated as follows: Starting from any current stage, an optimal policy for the rest of the process, i.e. for subsequent stages, is independent of the policy adopted in the previous stages. Of course, not all optimization problems can be presented as multistage decision processes for which the above principle is true. However, the class of problems for which it works is quite large.

If dynamic programming is applied to a combinatorial problem, then in order to calculate the optimal criterion value for any subset of size k, we first have to know the optimal value for each subset of size $k-l$. Thus, if our problem is characterized by a set of n elements, the number of subsets considered is 2^k. It means that dynamic programming algorithms are of exponential computational complexity. However, for problems which are NP-hard (but not in the strong sense) it is often possible to construct pseudopolynomial dynamic programming algorithms which are of practical value for reasonable instance sizes.

Branch and bound

Another method of implicit enumeration is used in so-called branch-and-bound algorithms. In this case, the enumeration is done by a tree search procedure, it means a procedure where the elements are nodes of a tree, which are explored in a special order. Each node is equivalent to a state of the DP algorithm and an arc corresponds to a decision. The ending node of an arc is equivalent to a final state of the DP algorithm.

A branch-and-bound procedure is an implicit enumerationmethod, characterized by two elements: the branching and the bound. The branching is the way the problem is decomposed. It is related to the definition of nodes and arcs. For example, a branching process can suppose that a sequence for a scheduling problem is developed from the first job to the last job of the schedule. It means that the root node of the tree is empty, the terminal nodes (leaves of the tree) are complete sequences, and each node is completed by one job in each of its child nodes.

Two sorts of bounds are considered. For a minimization problem, an upper bound UB is generally the value of a feasible solution, therefore greater than or equal to the value of an optimal solution. During the search process, this quantity is the value of the best known solution. A lower bound $LB(\sigma)$ is a quantity associated to a node σ of the tree. It is an under estimation of the value of the best solution which can be reached from this node, i.e. no feasible solution under this part of the tree has a solution less than this quantity. Therefore, if $LB(\sigma) \geq UB$, this part of the tree cannot help improving the best known solution value and there is no need to continue the enumeration of these solutions. Node σ is pruned or cutted and we can consider that this part of the tree has been implicitly explored.

The nodes can be explored using several procedures. The three most often used methods to explore the search tree are:

- The *breath first search* – nodes are managed in a first-in, first-out list.

- The *depth first search* – nodes are managed in a last-in,first-out stack.
- The *best first search* – nodes are sorted in non-decreasing order with respect to their lower bounds.

The algorithm stops when all the nodes have been implicitly explored.

1.4.2 Approximate methods

A heuristic or approximate method is an optimization method that aims to find a feasible solution of the objective function in a reasonable time, but without guarantee of optimality. The main advantage of these methods is that they can be applied to any class of problems,easy or very difficult. On the other hand the optimization algorithms such as simulated annealing algorithms, tabu algorithms and genetic algorithms have shown their robustness and efficiencies face several combinatorial optimization problems. The approximate methods include two classes:

- Constructive methods
- Metaheuristics

Constructive methods

Those are iterative methods that building a feasible solution step by step. Starting from an initially empty partial solution they search to extend the partial solution at every stage of the previous step,and this process is repeated until a complete solution is obtained. Constructive methods are generally used when quality solution is not a primary factor or the size of the instance is reasonable in this case for generating an initial solution we use metaheuristic, these are quick and easy implementation methods.
For example the Greedy algorithm (we will show this algorithm in the next chapter).

Metaheuristics

- Metaheuristics have been proposed which try to bypass these problems.
- Metaheuristics apply to solve the problems known as of difficult optimization
- Available from the 1980.

Definition 1.4.1. ***Metaheuristics***

- *A metaheuristic is a set of algorithmic concepts that can be used to define heuristic methods applicable to a wide set of different problems.*
- *A metaheuristic metaheuristic can be seen as a be seen as a general generalpurpose heuristic method toward promising regions of the search space containing high-quality solutions.*

- *A metaheuristic is a general algorithmic framework which can be applied to different optimization problems with relatively few modifications to make them adapted to a specific problem.*
- *Metaheuristics have capability to be extracted from a local minimum.*

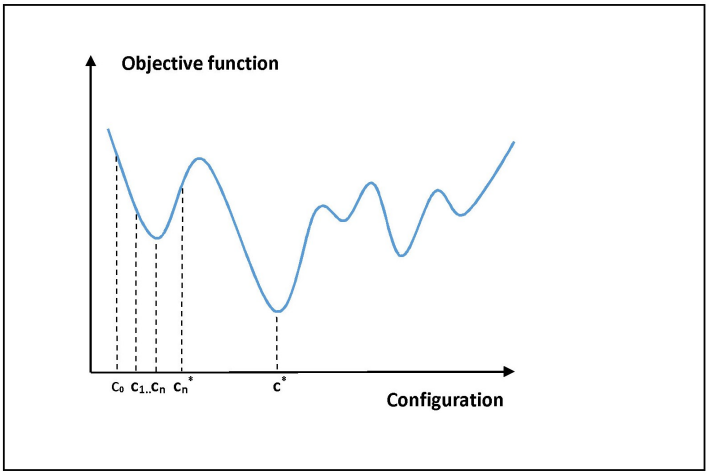

FIGURE 1.6 – Capability of metaheuristics

The metaheuristics are from now on regularly employed in all the sectors of engineering, Metaheuristics which are divided into two subclasses:

1. Single solution based metaheuristics methods.
2. Population methods.

1. **Single solution based metaheuristics methods**
   we present single solution based metaheuristics (S-metaheuristics) methods, also called trajectory methods. They could be viewed as walks (moves) through neighborhoods in the search space of the problem **[7]** S-metaheuristics methods are unlike P-metaheuristics methods, they iteratively apply the generation of the neighborhood solutions from the current single solution. This process iterates until a given stopping criteria e.g. (number of iterations). The most popular examples of such S-metaheuristics methods are tabu search (TS), simulated annealing (SA),variable neighborhood search (VNS), greedy randomized adaptive search procedure (GRASP) **[8]** ,**[9]** . The following subsections present a global overview of three method of S-metaheuristics i.e. tabu search, simulated annealing and variable neighborhood and their principles

(a) ***Tabu searsh***
In 1986, Glover proposed a deterministic method called tabu search method (TS) in order to escape from local optima **[10]** . In 1990s, tabu search method becomes very popular in solving optimization problems. The key feature of TS method is the use of memory, which records information related of the search process. TS generates a neighborhood solution from the current solution and accepts the best solution even if is not improving the current solution. This strategy may lead to cycles, i.e. the previous vistaed solutions could be selected again. In order to avoid cycles, TS discards the solution that have been previously visited by using memory which is called tabu list. The length of the memory (tabu list) control the search process. If the length of the tabu list is high the search will explore larger regions and forbids revisiting high number of solution. On the opposite, a low length of the tabu list concentrates the search on a small area of the search space. At each iteration the tabu list is updated (first in—first out queue). The tabu list contains a constant number of tabu moves called tabu tenure, which is the length of time for which a move is forbidden. If a move is good and can improve the search process but it is in tabu list, there is no need to be prohibited and the solution is accepted in a process called aspiration criteria. The main algorithm of tabu search method is reported in Algorithm 1. Good reviews of the TS method are provided in **[11]** ,**[12]** . TS have been applied to solve continuous optimization problems, see **[13]** ,**[14]** .

Algorithm 1 Tabu search algorithm

```
Set x = x_0;                                   ▷ Initial candidate solution
Set length(L) = z;                             ▷ Maximum tabu list length
Set L = ;                                      ▷ Initialize the tabu list
repeat
    Generate a random neighbor x';
    if x' ∉ L then
        if length(L) > z then
            Remove oldest solution from L;     ▷ First in first out queue
            Set x' ∈ L;
        end if
    end if
    if x' < x then
        x = x'
    end if
until Stopping criteria satisfied              ▷ e.g. Number of iterations
return x;                                      ▷ Best found solution
```

(b) ***Simulated Annealing*** Simulated annealing (SA) has been proposed by Kirkpatrick **[15]** , SA is probably the most widely used meta-heuristic in combinatorial optimization problem. It was motivated by the analogy between the physical annealing of metals and the process of searching for the optimal solution in a

combinatorial optimization problem. It is inspired by the Metropolis algorithm **[16]** . The main objective of SA method is to escape from local optima and so to delay the convergence. The basic SA algorithm is described as shown in Algorithm 2. SA proceeds in several iterations from an initial solution x^0. At each iteration, a random neighbor solution x' is generated. The neighbor solution that improves the cost function is always accepted. Otherwise, the neighbor solution is selected with a given probability that depends on the current temperature T and the amount of degradation ΔE of the objective function. ΔE represents the difference in the objective value between the current solution x and the generated neighboring solution x'. This probability follows, in general, the Boltzmann distribution as shown in Eq. 1.1.

$$P(\Delta E, T) = exp(\frac{-f(x') - f(x)}{T}) \tag{1.1}$$

Many trial solutions are generated as an exploration process at a particular level of temperature. The temperature is updated until stopping criteria satisfied.

Algorithm 2 Simulated annealing algorithm

```
Set x = x_0;                                          ▷ Generate initial solution
Set T = T_max;                                        ▷ startinig temperature
repeat
    repeat
        Generate a random neighbor x';
        ΔE = f(x') − f(x);
        if ΔE ≤ 0 then
            x = x';                                   ▷ Accept the neighbor solution
        else
            Accept x' with probability e^(ΔE/T) ;
            x = x';
        end if
    until Equilibrium condition   ▷ e.g. number of iterations executed at each T
    T = g(T);                                         ▷ Temperature update
until Stopping criteria satisfied                     ▷ e.g. T < T_min
return x;                                             ▷ Best found solution
```

In order to improve the performance of SA, we should carefully deal with the tuning of control parameters which included:

- **Choice of an initial temperature.** Choosing too high temperature will cost computation time expensively, while too low temperature will exclude the possibility of ascent steps, thus losing the global optimization feature of the method. We have to balance between these two extreme procedures.
- **Choice of the temperature reduction strategy.** If the temperature is decreased slowly, better solutions are obtained but with a more significant

computation time. On the other side, a fast decrement rule causes increasing the probability of being trapped in a local minimum.

- **Equilibrium State.** In order to reach an equilibrium state at each temperature, a number of sufficient transitions (moves) must be applied. The number of iterations must be set according to the size of the problem instance and particularly proportional to the neighborhood size.
- **Stopping criterion.** Concerning the stopping condition, theory suggests a final temperature equal to 0. In practice, one can stop the search when the probability of accepting a move is negligible, or reaching a final temperature T F.

2. **Population-based meta-heuristics techniques**
Population based metaheuristics methods (P-metaheuristics) start from an initial population of solutions, this is the main difference between them and the (S-metaheuristics) methods which start from a single solution. After the initial population is generated, the replacement phase is started by selecting a new population from the previous population. This operation iterates until a given stopping criteria. Most of the (P-metaheuristics) are nature-inspired methods. The most popular (P-metaheuristics) are evolutionary algorithms (EAs), particle swarm optimization (PSO), ant colony (AC). In the following we outline two of these methods,Which is :

   (a) **Evolutionary algorithms**
   Evolutionary algorithms (EAs) are stochastic (P-metaheuristics) that have been successfully applied tomany real and complex problems. EAs are based on the notion of competition. They are based on the evolution of a population of individuals, this population is usually generated randomly. Each individual in the population is evaluated by using an objective function (fitness function). At each generation, individuals with better fitness are selected to form the parents. Then the selected parents are reproduced using crossover and mutation operators to generate new offsprings. In the final stage a survival selection is applied to determine which individuals of the population will survive from the offsprings and the parents. This process is iterated until a stopping criteria are satisfied. Algorithm 4 illustrates the main steps of an evolutionary algorithm.

Algorithm 3 Evolutionary algorithms

```
Set the generation counter t = 0;
Generate an initial population P^0 randomly;          ▷ Initial population.
Evaluate the fitness function of all individuals in P^0;
repeat
    Set t = t+1;                                      ▷ Generation counter increasing.
    Select an intermediate population P^t from P^(t-1);   ▷ Selection operator.
    reproduced P^t ;                                  ▷ Crossover and mutation operators.
    Evaluate the fitness function of all individuals in P^t ;
until Termination criteria satisfied.
produce the best individual or best population found;
```

(b) **Genetic algorithm.**
A genetic algorithm is a computer algorithm that searches for good solutions to a problem from among a large number of possible solutions. All GAs begin with a set of solutions (represented by chromosomes) called population. A new population is created from solutions of an old population in hope of getting a better population. Solutions which are then chosen to form new solutions (offsprings) are selected according to their fitness. The more suitable the solutions are the bigger chances they have to reproduce. This process is repeated until some condition is satisfied [**17**] .
Most GAs methods are based on the following elements: "populations of chromosomes, selection according to fitness, crossover to produce new offspring, and random mutation of new offspring"[**14**] .

Algorithm 4 Genetic algorithm

1: Set the generation counter t = 0;
2: Generate an initial population P^0 randomly;
3: Evaluate the fitness function of all individuals in P^0;
4: **repeat**
5: Set t = t+1; ▷ Generation counter increasing.
6: Select an intermediate population P^t from P^{t-1}; ▷ Selection operator.
7: Associate a random number r from (0, 1) with each row in P^t;
8: **if** $r < P_c$ **then** ▷ P_c probability of crossover.
9: Apply crossover operator to all selected pairs of P^t ;
10: Update P^t ;
11: **end if** ▷ Crossover operator.
12: Associate a random number r_1 from (0, 1) with each gene ;
13: in each individual in P^t;
14: **if** $r < P_m$ **then**
15: Mutate the gene by generating a new random value for the selected ;
16: gene with its domain;
17: Update P^t ;
18: **end if**
19: Evaluate the fitness function of all individuals in P^t ;; ▷ Mutation operator.
20: **until** Termination criteria satisfied
21: produce the best individual or best population found;

1.5 Definition of the knapsack problem (KP)

The knapsack problem (KP) is a classic combinatorial optimization problem belonging to the class of NP-complete problems[**19**] . The statement of this problem is simple, it can be formally defined as follows: We are given an instance of the knapsack problem with item set N, consisting of n items j with profit p_j and weight w_j, and the capacity value c. (Usually, all these values are taken from the positive integer numbers). Then the objective is to select a subset of $N = 1, ..., n$ such that the total profit of the selected items is maximized and the total weight does not exceed c.

Alternatively, a knapsack problem can be formulated as a solution of the following linear integer programming formulation:

$$\text{maximize} \quad \sum_{j \in N} p_j x_j \tag{1.2}$$

$$\text{subject to} \quad \sum_{j \in N} w_j x_j \leq c \tag{1.3}$$

$$x_j \in \{0, 1\}, \quad j \in N, \tag{1.4}$$

- The variable x_i is the variable of decision; it takes the value of 1 if the object i is loaded in the bag, otherwise it takes the value of 0.

Without loss of generality, we assume that all the data are positive integers. In order to avoid trivial solutions, we assume that we have:

$$p_j, w_j \ \textit{and} \ c \ \textit{are positive integres}, \ j \in N. \tag{1.5}$$

$$\sum_{i=1}^{n} w_i > C \tag{1.6}$$

$$w_j < C \ \textit{for} \ j \in N. \tag{1.7}$$

we will always suppose that the items are ordered according to decreasing values of the profit per unit weight,i.e so that :

$$\frac{p_1}{w_1} \geq \frac{p_2}{w_2} \geq ... \geq \frac{p_j}{w_j} \tag{1.8}$$

We will denote the optimal solution vector by $x^* = (x_1^*, ..., x_n^*)$ and the optimal solution value by z^*. The set X^* denotes the optimal solution set, i.e. the set of items corresponding to the optimal solution vector.

Problem (KP) is the simplest non-trivial integer programming model with binary variables, only one single constraint and only positive coefficients. Nevertheless, adding the integrality condition (1.4) to the simple linear program (1.2)-(1.3) already puts (KP) into the class of "difficult" problems.

The knapsack problem has been studied for centuries as it is the simplest prototype of a maximization problem. Already in 1897 Mathews **[20]** , showed how several constraints may be aggregated into one single knapsack constraint. This is somehow a prototype of a reduction of a general integer program to (KP), thus proving that (KP) is at least as hard to solve as an integer program. It is however unclear how the name "Knapsack Problem" was invented. Dantzig is using the expression in his early work and thus the name could be a kind of folklore.

The interest in the knapsack is due to the fact that it allows modeling of very many problems such as the management problems of capital, loading of cargo,rotation of the crew, for touring and delivery; furthermore, this problem appears as a sub problem of many problems of combinatorial optimization.

Problems knapsack kind also appear frequently as a relaxation programming problems integer. In this type of application, it often resolves KP a problem to obtain an upper bound .We present in the following : Multiple knapsack problem and Multiple-choice knapsack problem. We also define an important concept is the "core" of the knapsack problem.

Continuous knapsack problem

The most natural, and historically the first, relaxation fo KP is *Linear Programming Relaxation*, i.e. *continuous knapsack problem* (CKP) obtained from (1.2),(1.3),(1.4) by removing the integrality constraint on x_j :

$$\begin{aligned} \text{maximize} \quad & \sum_{j\in N} p_j x_j \\ \text{subject to} \quad & \sum_{j\in N} w_j x_j \leq c \\ & 0 \leq x_j \leq 1, \quad j \in N, \end{aligned} \tag{1.9}$$

Suppose that the items,ordered according to (1.8), are consecutively inserted into the knapsack until the first item, s is found which does not fit. We call it the *critical item*, i.e

$$\min\{j : \sum_{i=1}^{j} w_i > c\}$$

and not that,because of assumptions (1.6),(1.7), we have $1 < s \leq n$. Then (CKP) can be solved through a property established by Dantzig(1957), which can be formally stated as follows.

Theorem 1.5.1. *An optimal solution vector x^* of **(CKP)** is:*

$$\begin{cases} x_k^* = 1, & k = 1, .., s-1 \\ x_s^* = \dfrac{1}{w_s}(c - \sum_{j=1}^{s} w_j), & k = s \\ x_k^* = 0, & k = s+1, .., n \end{cases}$$

The corresponding solution value is

$$z^{CKP} = \sum_{j=1}^{s-1} p_j + (c - \sum_{j=1}^{s-1} w_j)\frac{p_s}{w_s}$$

Proof. *See in* [**1**] .

1.6 Variants and Extensions of the Knapsack Problem

There are many variations of the knapsack problem that have arisen from the vast number of applications of the basic problem. The main variations occur by changing the number of some problem parameter such as the number of items, number of objectives, or even the number of knapsacks.

1.6.1 Subset sum problem

The subset sum problem is a special case of the decision and knapsack problems where each kind of item, the weight equals the value: $w_i = p_i$. This problem is NP-complete.

$$\text{maximize} \quad \sum_{j \in N} w_j x_j \tag{1.10}$$

$$\text{subject to} \quad \sum_{j \in N} w_j x_j \leq c \tag{1.11}$$

$$x_j \in \{0, 1\}, \quad j \in N, \tag{1.12}$$

The naive approach of computing the sum of the elements of every subset of S and then selecting the best requires exponential time. Below we present an exponential time exact algorithm

1.6.2 Multiple knapsack problem (MKP)

The multiple knapsack problem is a generalization of the standard knapsack problem (KP) from a single knapsack to m knapsacks with (possibly) different capacities.

Wen consider the problem where n given *items* should be packed m knapsacks of distinct *capacities* c_i, $i = 1, ..., n$. Each item j has an associated *profit* p_j and *weight* w_j, and the problem is to select m disjaoined subsets of items, such that subset i fite into knapsack i

and the total profit of the selected items is maximized. Thus we may formally define the Multiple knapsack problem (MKP) by

$$\text{maximize} \quad \sum_{i=1}^{m}\sum_{j=1}^{n} p_j x_{ij} \tag{1.13}$$

$$\text{subject to} \quad \sum_{j=1}^{n} w_j x_{ij} \leq c_i, \quad i = 1, ..., m \tag{1.14}$$

$$\sum_{j=1}^{m} x_{ij} \leq 1, \quad i = 1, ..., m \tag{1.15}$$

$$x_j \in 0, 1, i = 1, ..., m, j = 1, ..., n,$$

where $x_{ij} = 1$ if item j is assigned to knapsack j and $x_{ij} = 0$ otherwise.

There are sevral applications for MKP, as the problem directly reflects a situation of loading m ships/contaners or e.g. packing m envlopes. Martello and Toth [**21**] also proposed the problem used for deciding how to load m liquids into n tanks, when the liquids may not be mixed.
The Multiple knapsack problem is NP-hard in strong sense, and thus any DP approach would result in strictly expontial time bounds. Most od literature has thus been focused on Branch and Bound techniques, although Fischetti and Toth[**22**] used some kind of dominance tests to speed up the solution process.

1.6.3 Multiple-choice knapsack problem (MCKP)

The multiple-choice knapsack problem (MCKP) is a generalization of the ordinary knapsack problem, where the set of items is partitioned into classes. The binary choice of taking an item is replaced by the selection of exactly one item out of each class of items.
Given k classes $N_1, ..., N_m$of items to pack in knapsack of *capacity* c. Each item $j \in N_i$ has *profit* p_{ij} and a *weight* w_{ij}, and the problem is to choose one item from each class such that the ptofit sum is macximized without having the weight sum to exceed c.
The Multiple-choice knapsack problem (MCKP) may thus be formulated as :

$$\text{maximize} \quad \sum_{i=1}^{m}\sum_{j\in N_i} p_{ij} x_{ij} \tag{1.16}$$

$$\text{subject to} \quad \sum_{j=1}^{n}\sum_{j\in N_i} w_j x_{ij} \leq c \tag{1.17}$$

$$\tag{1.18}$$

$$\sum_{j\in N_i} x_{ij} = 1, \quad i = 1, ..., m \tag{1.19}$$

$$x_j \in 0, 1, \quad i = 1, ..., m, \quad j \in N_i,$$

All coefficients P_{ij}, w_{ij}, and c are positive integers, with class N_i having size n_i. The total number of items is $n = \sum_{i=1}^{m} n_i$.

MCKP is NP-hard as it conains KP as a special case, but it can be solved in pseudo-polynomial time through dynamic programming(Dudzinki and Walukiewicz 1987). The problem has a large range of applications.capital budgeting (Nauss 1978),Menu planning (Sinha and Zolterns 1979), transforming nonlinear KP to MCKP(Nauss 1978), Morever MCKP appear by Lagrange relaxation of sevral integer programming problems(Fisher 1981). Sevral algorithms for MCKP have been presented during the last two decades: e.g.Nauss 1978, Sinha and Zolterns 1979, and Dayer, Kayal and Walker 1984. Most of these algorithms start by solving LMCKP in order to obtain an upper bound for the problem.

1.6.4 The quadratic knapsack problem (QKP)

In all the variants of the knapsack problems considered so far the profit ofchoosing a given item was independent of the other items chosen. In many real life applications as well as in problems with roots in graph theory it is natural to assume that the profit of a packing also should reflect how well the given items fit together. One possible formulation of such an interdependence is the quadratic knapsack problem (QKP) in which an item has a corresponding profit and an additional profit is redeemed if the item is selected together with another item. (QKP) was first introduced by Gallo, Hammer and Simeone [**23**] and has been studied intensively in the last decade due to its simple structure and challenging difficulty. The problem can be formulated as follows:

$$\sum_{i \in N} \sum_{j \in N} p_{ij} x_i x_j \tag{1.20}$$

$$\sum_{j \in N} w_j x_j \leq c \tag{1.21}$$

$$x_j \in 0, 1, i = 1, ..., m, j \in N_i, \tag{1.22}$$

As opposed to most of the knapsack problems considered so far, (QKP) is NP-hard in the strong sense, which can be seen by reduction from the clique problem. As one might expect, due to its generality, (QKP) has a wide spectrum of applications. Witzgall [**24**] presented a problem which arises in telecommunications when a number of sites for satellite stations have to be selected, such that the global traffic between these stations is maximized and a budget constraint is respected. Similar models arise when considering the location of airports, railway stations or freight handling terminals[**25**] . Johnson, Mehrotra and Nemhauser [**26**] mention a compiler design problem which may be formulated as a (QKP), as described in [**27**] .

1.7 Conclusion

We defined in this chapter combinatorial optimization problems and Knapsack problem, in going from defining an optimization problem,passing through the definition of an optimal solution to a given problem. We mentioned some notins of algorithms and the complexity theory. Subsequently,we presented different approaches for solving combinatorial optimization problems. Then, we gave the definition of knapsack problem and their family. we will concentrate the next chapter in the following question: how to solve the knapsack problem, and the difference approach of these problem ?

Chapter 2

Methods of solving the knapsack problem

2.1 Introduction

In this chapter ,We will apply some solving methods of the knapsack problem,In the sub-section (2.2) We will present two exact methods used for solving thes problem: firstly, the brand and bound then the dynamic programmig. In the sub-section (2.3) We will also discuss how to compute upper bounds and lower for the knapsack problem. Therdore, we will present the variables reduction method which is a pretreatment step of a problem , to reduce its cardinality. Finally, we definitions the core of KP and how to find him.

2.2 Exact Solution of the Knapsack Problem

The Knapsack Problem is usually solved for exact results in one of two ways. The first method is called "branch and bound". The second method is known as "dynamic programming". There are additional possibilities for serial implementations for the Knapsack Problem. These include genetic algorithms and list splitting. Some of these, such as genetic algorithms, have been shown to perform worse than the traditional branch and bound or dynamic programming algorithms (Gordon, Boehm & Whitley 1994). Others are not applicable to parallel programming. Some algorithms, such as deux-list splitting, restrict how many sub-problems the problem can be divided into (Horowitz & Sahni, 1974). As such, they are not covered in this paper.

2.2.1 Branch and bound

The branch and bound method is a general method that permits one to find exact optimal solution of various optimization problems like discrete and combinatorial optimization problems. It is based on enumeration of all candidate solutions and pruning of large subsets of candidate solutions via computation of lower and upper bounds of the criterion to optimize (see Figure 2.1). We concentrate here on breadth-first search strategy. We assume that items are sorted according to decreasing profit per weight ratio.

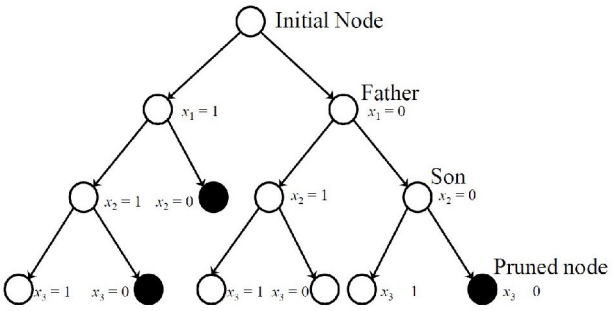

FIGURE 2.1 – Branch-decision tree

1. ***State notation***
   At each step of the branch and bound method, the same branching and bounding tasks are carried out on a list of states called also nodes. Let k denote the index of the current item relative to a branching step. We denote by N_e^k the set of items in the knapsack at step k for a given node e. A node e is usually characterized by a tuple $(w_e, p_e, X_e, U_e, L_e)$ where w_e represents the weight of node e, p_e represents the profit of the node, X_e is the solution subvector associated with node e ,U_e and L_e, respectively, are an upper bound and a lower bound of the node, respectively. We have:

$$w_e = \sum_{i \in N_e^k} w_i, \quad p_e = \sum_{i \in N_e^k} p_i \tag{2.1}$$

The so-called Dantzig bound (see [18]), derived from the solution of the continuous knapsack problem is a classical upper bound; it is given as follows:

$$U_e = p_e + \sum_{i=k+1}^{s_e-1} p_i + \lfloor \tilde{c}.\frac{p_{s_e}}{w_{s_e}} \rfloor \tag{2.2}$$

where s_e is the so-called slack variable such that

$$\sum_{i=k+1}^{s_e-1} w_i \leq c - w_e \leq \sum_{i=k+1}^{s_e} w_i \tag{2.3}$$

(we note that one can compute index s_e for a node e and item $k+1$ via a greedy algorithm applied to knapsack of capacity $c - w_e$) and $\tilde{c}$ is the residual capacity given by

$$\tilde{c} = c - w_e - \sum_{i=k+1}^{s_e-1} w_i \tag{2.4}$$

Classically, lower bound of a node can be obtained via a feasible solution of the knapsack problem with $\{0, 1\}$ variables. For example, a good lower bound L_e can be computed via a greedy algorithm and selection of all items after k and before slack variable, s_e,

since we have assumed that items are sorted according to decreasing price per weight ratio. Hence, we have:

$$L_e = p_e + \sum_{i=k+1}^{s_e-1} p_i + \sum_{i=s_e+1}^{n} p_i x_i \tag{2.5}$$

where $x_i = 1$ if $w_i \leq \sum_{i=s_e+1}^{i-1} w_i x_i$

For the sake of simplicity and efficiency, the following representation of a given node e will be used in the sequel: $(\tilde{w_e}, \tilde{p_e}, X_e, U_e, L_e)$, where

$$\tilde{w_e} = w_e + \sum_{i=k+1}^{s_e-1} w_i, \quad \textit{and} \quad \tilde{p_e} = p_e + \sum_{i=k+1}^{s_e-1} p_i \tag{2.6}$$
$$\textit{and} \quad L_e = \tilde{p_e} + \lfloor \tilde{c}.\frac{p_{s_e}}{w_{s_e}} \rfloor$$

and

$$L_e = \tilde{p_e} + \sum_{i=s_e+1}^{n} p_i x_i \tag{2.7}$$

where $x_i = 1$ if $w_i \leq \tilde{c} - \sum_{j=s_e+1}^{i-1} w_j x_j$

Obtaining this way the bounds U_e and L_e is more efficient since bound computation is time consuming in the branch and bound method.

2. ***Branch and bound procedure***:
   If $k < s_e$, then a node generates two sons at step k:

   - a node with $x_k = 0$, $\tilde{w_e} = \tilde{w_e} - w_k$ and $\tilde{p_e} = \tilde{p_e} - p_k$ in this case, one has to compute the new slack variable; moreover, the upper and lower bounds U_e and L_e, respectively, are updated according to the value of $\tilde{p_e}$.
   - a node with $x_k = 1$ (in this case, the son is similar to its father that is already in the list, in particular, the upper and lower bounds do not change; thus, no new node is created).

The case where $k = s_e$ yields only one son with $x_k = 0$, $x_k = 0$, $\tilde{w_e} = \tilde{w_e} - w_k$ and $\tilde{p_e} = \tilde{p_e} - p_k$, and $s_e = s_{e+1}$. Indeed clearly, the k-th item cannot be packed into the knapsack.

Pruning a subset of candidate solutions is then done via comparison of the best current lower bound with the upper bound of each node. We denote by $\bar{L}$ the best lower bound. If we have $U_e \leq \bar{L}$, then node e can be discarded.

Exemple 2.2.1. *Consider the instance of KP definded by* $n = 7 \quad c = 9$

j	1	2	3	4	5	6	7
w_j	6	5	8	9	6	7	3
p_j	2	3	6	7	5	9	4

The application of Branch-and-Bound to our example is illustrated in the branch-and-bound tree of Figure 2.2. Lower bounds are marked by underlined numbers and upper bounds by overlined numbers, respectively. The lower bounds are found by algorithm Greedy. The upper bounds are the values of U_{LP}, i.e. they are calculated by rounding down the solution of the linear programming relaxation (LKP). Of course, both bounds are recalculated depending on the variables which have been fixed already. We omitted branching when the corresponding item could not be packed into the knapsack since the capacity restriction was violated and proceed with the next node for which a packing of an item is possible. The bounds for the optimal solution are written in bold.
The optimaple solution of our example is $x^* = (1, 0, 0, 1, 0, 0, 0, 0)$

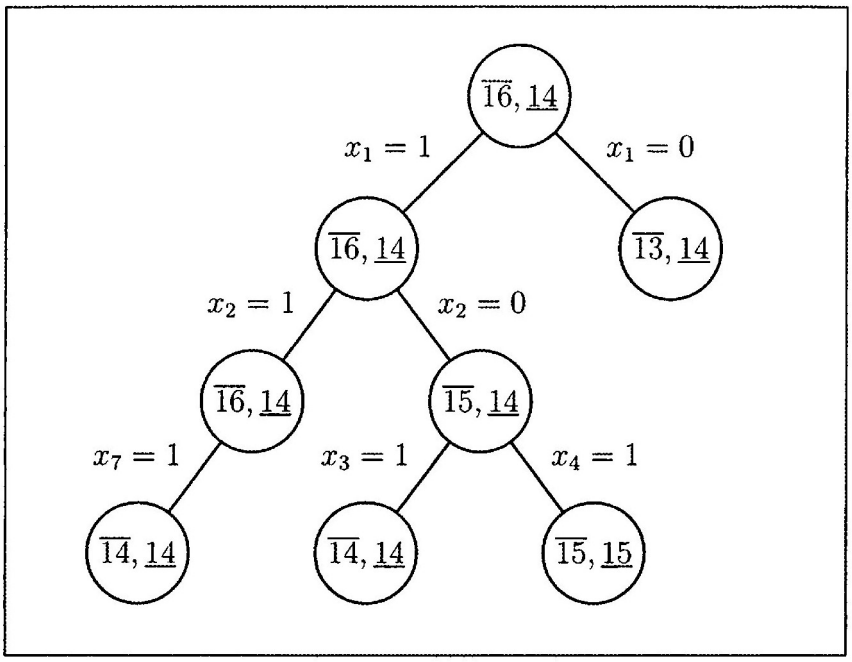

FIGURE 2.2 – A tree illustrating Branch-and-Bound applied to our example.

2.2.2 Dynamic Programming

Dynamic programming [**28**] ,[**5**] is a classic method of exact resolution that can be used to solve a large number of optimization problems whose knapsack problem. This is a recursive method.

We introduce the following concepts for step k, $k \in 1, ..., n$ where n is the number steps:

- X_k is the set of $\ll$ *decisions* $\gg$ that one takes in k,
- E_k is the set of $\ll$ *states* $\gg$ in which the system is to step k,
- $C(e_{k-1}, x_k)$ is the profit attached to the x_k decision that passes the system of e_{k-1} state to state e_k
- $f(k, e_k) = \sum_{j=1}^{k} C(e_{j-1}, x_j)$ is the total profit of the sequence of decisions
- $x = (x_1, x_2 x..., x_k)$ which makes the system pass of the state e_0 to the state e_k
- $F(e_{k-1}, x_k) = e_k$ is state transition.

Dynamic programming is based on the principles described below.

Principe 1 **Markov property**
The profit associated with the decision x_k when we are in the state e_{k-1} depends only on these two elements and not on all the previous decisions which allowed to end in the state e_{k-1}.

Principe 2 **The principle of optimality of Bellman**
Whatever is the optimal decision taken in the stage k which brings the system of the state ek-1 to the state ek the portion of the politics(policy) enters e0 and ek-1 is optimal.

Principe 3 It seeks an evolution of the global profit system (minimum or maximum) equal to the sum of the transition profits

Solving the problem is therefore to find an optimal sequence that $x^* = (x_1^*, x_2^*, ..., x_k^*)$ from the initial state e0 brings us to the state while maximizing the profit function:

$$f(k, e_n) = \max\{\sum_{j=1}^{k} C(e_{j-1}, x_j) | F(e_{j-1}, X_j); j \in \{1, .., n\}\} \tag{2.8}$$

Applying the principle of optimality, we can calculate by degrees $f(n, e_n)$ using the recursive equation as follows:

$$f(k, e_n) = \max_{x \in X_k; e_k = F(e_{k-1}, x_k)} \{C(e_{k-1}, x_k) + f(k-1, e_{k-1}), \} \tag{2.9}$$

where $f(k, e_k)$ is the optimal benefit of the problem over a horizon of k periods.

Dynamic programming for the knapsack problem

Considering the formulation of the problem given by KP (2.1), we define for each couple $(k, g), k = 1, ..., n, g = 0, \ldots, c$, the sub-problem (KP(k,g)):

$$(KP(k,g)) : \begin{cases} \max \sum_{j=1}^{k} p_j x_j \\ s.c. \sum_{j=1}^{k} w_j x_j \leq g \\ x_j \in \{0,1\}, j \in \{1, ..., n\} \end{cases} \tag{2.10}$$

The profit $f(k, g)$ is obtained by loading the backpack capacity g with the first k objects. Hence the solution of the KP problem is given by the solution of (KP (n,c))

In this problem we have to step k:

- All decisions :
$$X_k = \{0, 1\} \tag{2.11}$$
- All states in which the system can be:
$$E_k = \{e_k = g | g \in \{0, 1, ..., \min\{c, \sum_{j=1}^{k} w_j\}\}\} \tag{2.12}$$
- The profit of the decision x_k which makes the system pass of the state e_{k-1} to the state e_k
$$C(e_{k-1,x_k} = p_k x_k \tag{2.13}$$
- State transition:
$$F(e_{k-1,x_k}) = e_k = e_{k-1} + w_k x_k \tag{2.14}$$

The total profit of the sequence of decisions $x = (x_1, x_2 x..., x_k)$ which makes the system pass of the state e_0 in e_k:

$$\sum_{j=1}^{k} C(e_{j-1}, x_j) = \sum_{j=1}^{k} p_j x_j \tag{2.15}$$

The list of the dynamic programming in the stage k is constituted by the following elements:

- the set E_k of states in which the system can be found in k,
- The sequence of optimal decisions $x_g = (x_{g,1}, x_{g,2} x..., x_{g,k})$ which maximizes the profit for a knapsack of capacity g.

We then have for each $g \in \{1, 2, ..., n\}$,define the following function:

$$f(k, g) = \sum_{j=1}^{k} C(e_{k-1}, x_{g,j}) \tag{2.16}$$

The relation which connects all the elements of the list of the dynamic programming with the stage k, with the elements of the list in the stage k - 1,allows us to build recursively list containing all the solutions of the problems f (k, g) for a fixed k:

$$f(k,g) = \begin{cases} f(k-1,g) \quad pour \quad g \in \{0,1,.....,w_k-1\} \\ \max\{f(k-1,g), c_k + f(k-1,g-w_k \quad pour \quad g \in \{w_k,...,c^*\}\} \end{cases} \tag{2.17}$$

$and \quad c^* = \min\{c, \sum_{j=1}^{k} w_j\}$

Algorithm 5 DP algorithm

```
1: for g = 0 : c do
2:     f(0, g) = 0                                          ▷ initialization.
3: end for
4: for k = 1 : n do
5:     for g = 0 : w_j − 1 do                               ▷ item k is too large to be packed.
6:         f(k, g) = f(k − 1, g)
7:     end for
8:     for g = w_k : c do                                   ▷ item k may be packed
9:         if (f(k − 1, g − w_k) + p_k) > f(k − 1, g) then
10:            f(k, g) = f(k − 1, g − w_k) + p_k
11:        else f(k, g) = f(k − 1, g)
12:        end if
13:    end for
14: end for
15: i=n, k=W                                                ▷ find actual Knapsack Items
16: while i do,k>0
17:    if B[i, k] ≠ B[i − 1, k] then
18:        mark the i^th item as in the knapsack            ▷ Finding the Items
19:        i=i-1,k=k-w_j
20:    elsei=i-1
21:    end if
22: end while
```

Exemple 2.2.2. *Let's run our algorithm on the fllowing data: n = 4 (of elements) W = 5 (max weight)*

j		1	2	3	4
$Weight$	w_j	2	3	4	5
$Benifit$	p_j	3	4	5	6

The execution of algorithm DP give the following table :

n \ W	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4	4	7
3	0	0	3	4	5	7
4	0	0	3	4	5	7

TABLE 2.1 – The results of the execution of the (DP)

Comments

- *All of the information we need is in the table.*
- *This algorithm only finds the max possible value that can be carried in the knapsack i.e., the value in B[n,W]*
- *To know the items that make this maximum value, an addition to this algorithm is necessary.*
- *The optimal knapsack should contain* $\{1, 2\}$

The concept lists

At each stage k, we update a list which is defined as follows:

$$L_k = \{(w,p) | w = \sum_{j=1}^{k} w_j x_j \leq c, p = \sum_{j=1}^{k} p_j x_j \quad \forall j \in \{1,..,k\} \quad x_j \in \{0,1\}\} \tag{2.18}$$

It follows from the dynamic programming principle that the use of the concept of dominated states permits one to reduce drastically the size of lists L_k since dominated states can be eliminated from the list with no loss for the solution of KP(g,k):

Dominated state: Let (w, p) be a couple of weight and profit, i.e. a state of the problem. If $\exists(w', p')$ such that $w' \leq w$ and $p' \geq p$, then (w, p) is dominated by (w', p').

Exemple 2.2.3. *Consider the instance of (KP) defined by : n = 4 , c = 15, objets [(3,2), (5,4), (8,7), (10,10)]*

$$P = \begin{cases} \max 2.x_1 + 4.x_2 + 7.x_3 + 10.x_4 \\ 3.x_1 + 5.x_2 + 7.x_3 + 10.x_4 \leq 15 \\ x_j \in \{0,1\} \quad \forall j \in \{1,..,4\} \end{cases} \tag{2.19}$$

The resolution of the problem (2.19) via the method of dynamic programming with lists, is shown in Table 2.2.
The principle of dominance stems directly from the principle of optimality.Therefore dominated the items are removed from the list of dynamic programming, thus considerably reducing the size of lists and therefore the research of an optimal solution.

Table 2.2 shows the removal of elements dominated (in gray) in the dynamic programming list.

In the example above, the pair (11.9) corresponding to the combination of items 1 and 3, is dominated by the pair (10,10) because it has a higher weight but lower profit.

L0		L1		L2		L3		L4	
(0, 0)		(3,2)		(5,4)		(8,7)		(10,10)	
w	*P*	*w*	*p*	*w*	*p*	*W*	*p*	*w*	*p*
								15	14
								13	12
								13	11
						13	11	11	9
						11	9	10	10
						8	7	8	7
				8	6	8	6	8	6
				5	4	5	4	5	4
		3	2	3	2	3	2	3	2
0	0	0	0	0	0	0	0	0	0

TABLE 2.2 – Lists of dynamic programming (in gray, eliminated states)..

The dynamic programming algorithm is equal in complexity to $O(\min(2^n, n.c))$, (Ahrens and Finke [**29**]) , and Horowitz and Sahni [**30**] .

Algorithm 6 DP-with List algorithm

1: $L_0 = \langle (0,0) \rangle$
2: **for** $j = 1 : n$ **do**
3: $\quad L'_{j-1} = L_{j-1} \bigoplus (w_j, p_j)$ ▷ add (w_j, p_j) to all elements in L_{j-l}.
4: $\quad$ delete all states $(\bar{w}, \bar{p}) \in L'_{j-1}$ with $\bar{w} > c$
5: $\quad L_j = \quad MergeLists(L_{j-1}, L'_{j-1})$
6: **end for**
7: return the largest state in L_n

Decomposition of the problem (method two lists)

Horowitz and sahni (1974) presented an algorithm based on the subdivision of the original problem of n variables into two subproblems (KP1 and KP2), respectively of q = [n/2] and r = n-q variables.

$$(KP1) : \begin{cases} \max \sum_{j=1}^{q} p_j x_j \\ s.c. \sum_{j=1}^{q} w_j x_j \leq c \\ x_j \in \{0,1\}, j \in \{1, ..., q\} \end{cases} \quad and \quad (KP2) : \begin{cases} \max \sum_{j=q+1}^{n} p_j x_j \\ s.c. \sum_{j=q+1}^{n} w_j x_j \leq c \\ x_j \in \{0,1\}, j \in \{q+1, ..., n\} \end{cases}$$

For each subproblem a list containing all the undominated states relative to last stage is computed, the two lists are then combined in order to find the optima solution.
we denote the two lists L1 and L2.

Solve the initial problem (KP) back to search among all possible combinations achievable between the states of the two lists, one that maximizes profit. That is to merge the two lists. For this research, the L1 list is traversed by the decreasing weight and L2 according to the increasing weight. It uses a pivot P to be alternately in L1 and then L2.

P = (w, p)being in L1,L2 until one traverses (w', p') such that $w + w' \leq c$. The new pivot is then P = (w', p'). Then continued the golf of list L1 from (w', p') to (w",p") as $w' + w" \leq c$, and takes P = (w",p") and so on. The search stops when the end of one of the two lists is reached.

The time complexity construction of the two lists is $O(\min(2^{\frac{n}{2}}, \frac{n}{2}.c))$, and the merger of the two lists is $O(\min(2^{\frac{n}{2}}, c))$.Regarding the spatial complexity,it passes from $O(\min(2^n, c))$ to $O(\min(2^{\frac{n}{2}}, c))$

Temporal complexity and the spatial complexity of the dynamic programming algorithm, thus find themselves reduced.

2.3 Terminals of calculation and variable reduction

It is obvious from the description of the method for Branch and Bound as calculating upper and lower bounds is an important point to refine the performance of this method.
We will briefly review some of the bounding methods given in the literature. Most of them are based on the relaxations introduced in section (2.2).

2.3.1 Upper bound

Most of the upper bounds for (KP) rely on some kind of sorting according to the efficiency e_j of an item j defined as $e_j = \frac{p_j}{w_j}$. For simplicity of the presentation we will in the following assume that the items are sorted according to decreasing efficiencies

$$\frac{p_1}{w_1} \geq \frac{p_2}{w_2} \geq ... \geq \frac{p_j}{w_j} \tag{2.20}$$

although most of the bounds can be derived faster without this sorting.

The simplest upper bound for (KP) can be derived by relaxing the constraint $x_1 \in \{0, 1\}$ on the most efficient item to the weaker constraint $x_1 \geq 1$. This leads to the trivial bound

$$U_0 = \lfloor \frac{cp_1}{w_1} \rfloor \tag{2.21}$$

The LP-relaxation (LKP) of (KP) as introduced in Subsection (2.2,2) leads to the so-called Dantzig bound **[31]** which may be derived by use of Theorem (2.2.2). As in (2.50), we may round down the objective of the LP-relaxation thus getting the bound

$$U_1 = U_{LKP} = \lfloor Z^{LKP} \rfloor$$
$$where \quad Z^{LKP} = \sum_{j=1}^{s-1} p_j + (c - \sum_{j=1}^{s-1} w_j)\frac{p_s}{w_s}$$

We observe that $U_1 \leq U_0$ and hence U_1 may be derived in a straightforward way in $O(n)$ time.

A different bound can be obtained through Lagrangian Relaxation. For the knapsack problem we can only relax the single capacity constraint with a nonnegative

$$L(KP,\lambda) : \left\{ \begin{array}{ll} \texttt{maximize} & \sum_{j=1}^{n} p_j x_j + \lambda(c - \sum_{j=1}^{n} w_j x_j) \\ \texttt{subject to} & x_j \in \{0,1\} \qquad\qquad j = 1, ...n. \end{array} \right. \tag{2.22}$$

Indeed a bound equal to that of the LP-relaxation may be obtained by choosing $\lambda = \frac{p_s}{w_s}$. Hence the best bound obtained by $L(KP,\lambda)$ equals U_1 **[32]**

Considering separately the case of packing the split item s or not, Martello and Toth **[33]** derived the following upper bound

$$U_2 = \left\{ \lfloor \sum_{j=1}^{s-1} p_j + c - \sum_{j=1}^{s-1} w_j \frac{p_{s+1}}{w_{s+1}} \rfloor, \lfloor \sum_{j=1}^{s-1} p_j + p_s + c - \sum_{j=1}^{s-1} w_j - w_s \frac{p_{s-1}}{w_{s-1}} \rfloor \right\} \tag{2.23}$$

It is easy to see that U_2 can be computed in $O(n)$ time as we only need to know $s-1, s$ and $s+1$. Moreover, we observe that $U_2 \leq U_1$.

Generalizing this principle, Martello and Toth **[33]** proposed the derivation of arbitrarily tight upper bounds by using *partial enumeration*. Assume that the items are sorted according to decreasing efficiencies (2.20), let $M \subseteq \{1, ..., n\}$ be a subset of the items and assume that $X_M = \{x_j \in \{0,1\}, j \in M\}$. Then every optimal solution of (KP) must have its origin in one of the zero-one vectors in X_M and an upper bound on (KP) is thus given by

$$U_M = \max_{\tilde{x} \in X_M} U(\tilde{x}) \tag{2.24}$$

where $U(\tilde{x})$ is any upper bound on (KP) with the additional constraint $x_j = \tilde{x_j}$ for $j \in M$. Since the computational effort to compute U_M is $O(2^{|M|})$ times the complexity of deriving $U(\tilde{x})$, this approach is only useful for small sets M.

Other more or less straightforward bounds along similar or related lines were derived by Fayard and Plateau[**34**] , Miiller-Merbach [**35**] , and Dudzinski and Walukiewicz [**36**] .

2.3.2 Lower Bounds for (KP)

In order to solve the (KP) through branch-and-bound. A lower bound L is needed which the upper bounds can be tested against. Although one may initially set $L = 0$ good branch-and-bound implementations rely on being able to construct a lower bound close to the optimal solution.
The simplest lower bound can be obtained by Greedy algorithm (see in sectoin 2.4.1)as follows:

$$L = \max_{i=s+1,..,n} \left\{ \sum_{j=s+1}^{n} p_j + p_i \;\mid\; \sum_{j=s+1}^{n} w_j + w_i \leq c \right\} \tag{2.25}$$

There exist other lower bound has been proposed by Pisinger, So the article in the index s is loaded and apply the algorithm Greedy for fill the knapsack of capacity $c - w_s$.Thus the lower bound obtained is as follows:

$$L^{'} = \max_{i=1,..,s-1} \left\{ \sum_{j=1}^{s-1} p_j - p_i \;\mid\; \sum_{j=1}^{s-1} w_j - w_i \leq c \right\} \tag{2.26}$$

The time complexity of bounds $L, L^{'}$ is $O(n)$ but the performance ratio is arbitrarily close to zero.
Several of the approximation techniques can also be used to derive a lower bound L for (KP). however at the cost of a higher computational cost.

2.3.3 Variable Reduction

It is clearly advantageous to reduce the size of a (KP) before it is solved. This holds in particular if the problem should be solved to optimality, but also heuristics and approximation algorithms will in general benefit from a smaller instance.

Variable reduction considerably reduces the observed running time of algorithms for nearly all instances occurring in practice, although it does not decrease the theoretical worstcase running time as instances may be constructed where the reduction techniques have no effect.

Variable reduction algorithms may be seen as a special case of the branch-andbound paradigm where we only consider branching on a single variable at the root node. All variables are in turn chosen as the branching variable at the root node, and different branches are investigated corresponding to the domain of the variable.

Considering (KP) we can branch on every variable x_j with $j = 1, ..., n$ yielding two subproblems as depicted in Figure(2.1).

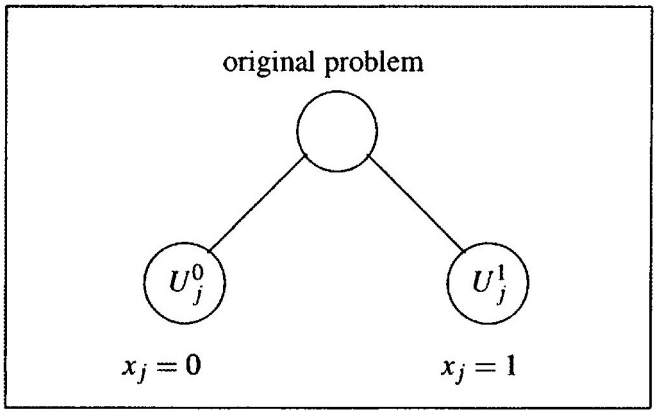

FIGURE 2.3 – Branching on the two possible solution values of x_j.

Let U_j^0 be an upper bound for (KP) with additional constraint $x_j = 0$. Similarly let U_j^1 be an upper bound for (KP) with constraint $x_j = 1$. Moreover, assume that an incumbent solution value z has been found in some way. If $U_j^0 \leq z$, then we know that the branch $x_j = 0$ does not lead to an improved solution and thus we may fix the variable to $x_j = 1$. In a similar way $U_j^1 \leq z$ implies that $x_j = 0$ in every improved solution. All variables fixed at their optimal value may be removed from the problem thus decreasing the size of the instance.
This is important as the reduction attempts to fix variables at their optimal value in every improved solution. If no improved solution is found, one should go back to the last incumbent solution.

Ingargiola and Korsh [**38**] presented the first reduction algorithm for the (KP). But several improvements have emerged during the years.
For the following discussion assume that a lower bound z^l has been found through any kind of heuristic and that the corresponding solution vector x has been saved.
The number of binary variables (x_j) and the value of c can be decreased by applying reduction procedures which fix the optimal value of as many variables as possible. These procedures partition set $N = \{1, 2, ..., n\}$ three subsets:

$$\begin{aligned} N_1 &= \{j \in N : x_j = 1 \quad \textit{in an optimal solution to KP } \}; \\ N_0 &= \{j \in N : x_j = 0 \quad \textit{in an optimal solution to KP } \}; \\ F &= N - (N_1 \cup N_0). \end{aligned} \tag{2.27}$$

If $F = \emptyset$ or if $\sum_{j \in N_1} w_j > c$ then z^l is optimal and we may terminate.

The original KP can now be transformed into the reduced form

$$\begin{aligned} \text{(KPR)} \qquad \text{maximize} \quad & \sum_{j \in F} p_j x_j + \tilde{p} \\ \text{subject to} \quad & \sum_{j \in F} w_j x_j \leq c - \tilde{w} \\ & x_j \in 0, 1, i = 1, ..., m, j \in F, \end{aligned} \tag{2.28}$$

where $\tilde{p} = \sum_{j \in N_1} p_j, \quad \tilde{w} = \sum_{j \in N_1} w_j$

The optimal solution value is found as $z^* = \max z^l, z(KPR)$, z^l where is the lower bound used in the reduction.Notice, that for most bounds there is no reason for deriving U_j^1 for $j < s$ since the additional constraint $x_j = 1$ will not affect the bound, and hence it will not be possible to reduce any variables. Similarly, deriving U for $j > s$ is obsolete for bounds appearing by linear relaxation.

The time complexity of the reduction depends on the complexity of the applied bound. Dembo and Hammer used the following bounds which can be derived in constant time once the split item s has been found

$$U_j^0 = \tilde{p} - p_j + (c - \tilde{w} + w_j)\frac{w_s}{w_s} \qquad U_j^1 = \tilde{p} + p_j + (c - \tilde{w} + w_j)\frac{w_s}{w_s} \tag{2.29}$$

Both bounds appear by relaxing the constraint $x_s \in \{0, 1\}$ such that x_s becomes an unbounded (positive or negative) real variable. Using these bounds the whole reduction runs in $O(n)$ time.

2.3.4 The Core Concept

The one-dimensional 0/1 knapsack problem (KP) considers items $j = 1, ..., n$, associated profits p_j , and weights w_j . A subset of these items has to be selected and packed into a knapsack having a capacity c. The total profit of the items in the knapsack has to be maximized, while the total weight is not allowed to exceed c. If the items are sorted according to decreasing efficiency values

$$e_j = \frac{p_j}{w_j} \tag{2.30}$$

it is well known that the solution of the **LP-relaxation** consists in general of three consecutive parts: The first part contains variables set to 1, the second part consists of at most one split item s, whose corresponding **LP-values** is fractional, and finally the remaining variables, which are always set to zero, form the third part. For most instances of KP (except those with a very special structure of profits and weights) the integer optimal solution closely corresponds to this partitioning in the sense that it contains most of the highly efficient items of the first part, some items with medium efficiencies near the split item, and almost no items with low efficiencies from the third part. Items of medium efficiency constitute the so called core.

Balas and Zemel gave the following precise definition of the core of a one-dimensional 0/1 knapsack problem, based on the knowledge of an optimal integer solution x^*.

Definition 2.3.1. *Assume that the items are ordered according to decreasing efficiencies as in (2.5) and let an optimal solution vector be given by* x^* *. Define*

$$a := \min\{j \mid x_j^* = 0\}, \qquad b := \max\{j \mid x_j^* = 1\}. \tag{2.31}$$

The core is given by the items in the interval $C = a, ..., b$. *It is obvious that the split item is always part of the core.*

The KP Core (KPC) problem is defined as

$$\text{maximize} \sum_{j \in C} p_j x_j \tag{2.32}$$

$$\text{subject to} \quad \sum_{j \in C} w_j x_j \leq c - \tilde{w} \tag{2.33}$$

$$x_j \in \{0, 1\}, \quad j \in C, \tag{2.34}$$

with $\tilde{p} = \sum_{j=1}^{a-1} P_j$ *and* $\tilde{w} = \sum_{j=1}^{a-1} w_j$.

The solution of KPC would suffice to compute the optimal solution of KP, which, however, has to be already partially known to determine C.Pisinger [**37**] reported experimental investigations of the exact core size. He moreover studied the hardness of core problems, giving also a model for their expected hardness in [**33**] .

The first class of core algorithms is based on solving a core problem with an approximate core of fixed size $c = s - \delta, ..., s + \delta$ with various choices of δ, e.g. with δ being a constant or $\delta = \sqrt{n}$. An example is the MT2 algorithm by Martello and Toth [**33**] .

2.4 Approximate methods of Knapsack Problem

As for most NP-complete problems, it may be enough to find workable solutions even if they are not optimal. Preferably, however, the approximation comes with a guarantee on the difference between the value of the solution found and the value of the optimal solution. As with many useful but computationally complex algorithms, there has been substantial research on creating and analyzing algorithms that approximate a solution. The knapsack problem, though NP-Hard, is one of a collection of algorithms that can still be approximated to any specified degree. This means that the problem has a polynomial time approximation scheme. In this sub-section we have three approximate methods, it is algorithme greedy,polynomial time approximation scheme (PTAS) and fully polynomial time approximation scheme (FPTAS).

2.4.1 The Greedy Algorithm

If a non-expert were trying to find a good solution for the knapsack problem (KP), i.e. a profitable packing of items into the knapsack, an intuitive approach would be to consider the profit to weight ratio e j of each item which is also called the efficiency of an item with

$$e_j = \frac{p_j}{w_j}$$

and try to put the items with highest efficiency into the knapsack. Clearly, these items generate the highest profit while consuming the lowest amount of capacity. Therefore, according to assumation in (1.8).
The idea of the greedy algorithm with solution value z^G is to start with an empty knapsack and simply go through the items in this decreasing order of efficiencies adding every item under consideration into the knapsack if the capacity constraint (1.2) is not violated thereby. An explicit description of this algorithm Greedy is given in Figure 2.1.

Algorithm 7 Greedy algorithm

```
1: w̄ = 0;                          ▷ w̄ is the total weight of the currently packed items
2: Set z^G = 0;                     ▷ zG is the profit of the current solution
3: for j=1:n do
4:     if w̄ + w_j ≤ c then
5:         X_j = 1                  ▷ put item j into the knapsack
6:         w̄ = w̄ + w_j
7:         z^G = z^G + P_j
8:     elseX_j = 0
9:     end if
10: end for
```

A slight variation of Greedy is algorithm Greedy-Split which stops as soon as algorithm Greedy failed for the first time to put an item into the knapsack. After sorting the items according to (2.5) in O(nlogn) time the running time both of Greedy and Greedy-Split is O(n) since every item is considered at most once.

Exemple 2.4.1. *We consider an instance of (KP) with capacity c = 9 and n = 7 items with the following profit and weight values:*

j	1	2	3	4	5	6	7
w_j	6	5	8	9	6	7	3
p_j	2	3	6	7	5	9	4

TABLE 2.3 – Simple knapsack instance.

The items are already sorted by their efficiency in decreasing order. Algorithm Greedy puts items 1,2 and 7 into the knapsack yielding a profit z^G = 14, whereas Greedy-Split stops after assigning items 1 and 2 with total profit 11. The optimal solution X^* = 1,4 has a solution value of 15.

2.4.2 Polynomial time approximation scheme (PTAS)

The first approximation scheme fo KP was proposed by sahni(1975)and makes use of greedy-type procedure which finds a heuristic solution by fillin, in order of decreasing p_j/w_j

ratios, that part of c which is left vacant after the items of a given set M have been put into the knapsack. Given $M \subset N$ and assumin that the items are sorted according to (2.7), the procedure is as follows.

Algorithm 8 Greedy-sahni

Input $n, c, (p_j), (w_j), M$;
Output z^G, X;
begin
$z^G = 0$;
$\tilde{c} = c - \sum_{j \in M}$;
$X = \emptyset$;
for j=1:n **do**
    if $j \notin M$ *and* $w_j \leq \tilde{c}$ **then**
        $z^G = z^G + P_j$
        $\tilde{c} = \tilde{c} - w_j$
        $X = X \cup \{j\}$
    end if
end for
end

Given a non-negative integer parameter k, the sahni scheme $S(k)$ is

Algorithm 9 The sahni scheme S(k)

Input $n, c, (p_j), (w_j), M$;
Output z^h, X^h;
begin
$z^h = 0$;
for each $M \subset \{1, .., n\}$ *such that* $|M| \leq k$ *and* $\sum_{j \in M} w_j \leq c$;
begin;
call GS
if $z^G + \sum_{j \in M} P_j > z^h$ **then**
    $z^h = z^h + \sum_{j \in M} P_j$
    $X = X \cup \{M\}$
end if
end

Since the time complexity of procedure GS is $O(n)$ and the number of items it is exexuted is $O(n^k)$, the tme complexity of $S(k)$ is $O(n^{k+1})$, the space complexity is $O(n)$.

2.4.3 Fully polynomial time approximation scheme (FPTAS)

The fully polynomial time approximation scheme (FPTAS) for the knapsack problem takes advantage of the fact that the reason the problem has no known polynomial time solutions is because the profits associated with the items are not restricted. If one rounds off some of the least significant digits of the profit values then they will be bounded by a polynomial

and $1/\varepsilon$ where ε is a bound on the correctness of the solution. This restriction then means that an algorithm can find a solution in polynomial time that is correct within a factor of $(1-\varepsilon)$ of the optimal solution. An Algorithm for FPTAS

Algorithm 10 FPTAS

1: **Input** $\varepsilon \in [0,1]$;
2: a list A of n items, specified by their values,v_j, and weights
3: **Output** $s^{'}$ the FPTAS solution;
4: begin
5: $P = \max\{v_j | 1 \leq i \leq n\}$; ▷ the highest item value
6: $k = \varepsilon P/n$;
7: **for** j=1:n **do**
8: $v_i^{'} = \lfloor v_i/k \rfloor$
9: **end for**
10: **return** the solution, $s^{'}$, using the v_i values in the dynamic program outlined above
11: end

2.4.4 Knapsack problem by simulated annealing

simulated annealing: is when we start with a large chance of going down and then decrase probability as the algorithme goes on. The name is in analogy with of annealing of mental where a high temperature lets the atoms move around and fill in holes in the crystal structure, and controlled cooling maintains this as the atoms calm dwon.
For simulated annealing suupose we are at X, let $Y \in N(X)$ be a feasible solution returned by the neighbourhood search. If $P(Y) > p(X)$ then move to Y, otherwise with probability $\exp(P(Y) - P(X))/T$ move to Y, where T decrease by $T = \alpha T$ at each iteration $0 < \alpha < 1$. The generic shape of algorithm was given in chapter.

To use simulated annealing for the knapsack problem make the following choices

- $N(X) = \{X \in 0,1n : d(X,Y) = 1\}$ where d is the Hamming distance, for example :

  for $x = 01101$, $N(x) = \{01100, 01111, 01001, 00101, 11101\}$
  Neighborhood size of a binary string of length N.

- Given X, generate a random $Y \in N(X)$ by choosing a random index $0 \leq j \leq n-1$ and swapping that bit. Then

$$w(Y) : \begin{cases} w(X) + w_j & if \quad x_j = 0 \\ w(X) - w_j & if \quad x_j = 1 \end{cases}$$

  and

$$P(Y) - P(X) = \begin{cases} p_j & if \quad x_j = 0 \\ -p_j & if \quad x_j = 1 \end{cases}$$

- begin with the feasible solution (0,0, . . . ,0).

This gives the following algorithm

Algorithm 11 Simulated annealing knapsack

```
1: c = 0 ,T = T0 , X = [0,...,0], CurW = 0; bestX = X ;
2: while c ≤ cmax do
3:     j random integer 0 ≤ j ≤ n − 1
4:     Y = X; y(j) = 1-x(j);
5:     if not(y(j) = 1 and CurW + w(j)>M then▷ (the condition inside the not is the fail)
6:         if y(j) = 1 then                                  ▷ (that is if the profit increased)
7:             X = Y; CurW = CurW + w(j);
8:             if P(X) > P(bestX) then
9:                 bestX = X
10:            else r = rand(0,1)
11:            end if
12:            if r < exp((P(Y) − P(X))/T) then
13:                X = Y; CurW = CurW - w(j)
14:            end if
15:        end if
16:    end if
17: end while
18: c = c+1
19: T = alpha*T
20: return bestX
```

2.5 Conclusion

We have presented in this chapter some methods of resolution of the Knapsach problem, these methods have been detailed in this chapter.and was given some examples in the next chapter we will execute the algorithms of the chapter in programming compiler, and we will generate the results in data, and give the comparison about the algorithms.

Chapter 3

Computational experiments

3.1 Introduction

We show in this chapter how our algorithms in the previous chapter can be applied to the knapsack problem. We test and compare the variants described in the previous chapter on instances. We also compare between the approximate algorithms and exact algorithms.

3.2 Programming and implementation

The implementation phase is usually characterized by a set of choices made by the designer or programmer to respond to a set of related criteria to the proposed method, in our case we will specify the programming language; consequently we will designer the tools or platform programming and implementation fits the selected language. Other choices related to the method and must be justified, which is the data structure supporting a solution, supporting the local memory and global memory, and the choice of neighborhood-type dune solution.

3.2.1 The choice of programming language

There are several programming languages to implement mathematical algorithms, but for portability reasons the method has been implemented with MATLAB, MATLAB portability gives it an advantage to be a good implementation language for applications on mathematical algorithms.
Generally, Matlab is used to experiment calculation very quickly. Some programs that require 1 day of programming in C / C ++ can be achieved in 1 hour Matlab. By against, once programmed, the calculation time in Matlab can be 100 times higher than that of C / C ++. Therefore, it is only used very little to achieve a finished product for individuals.

The MATLAB language provides several mechanisms to implement a given program to be operated, and facilitates handling structures (derived type object), offering the following possibilities:

- Infinitely faster programming to calculate and display.
- A rich library.
- Ability to include a program in C/C ++.

- Interpreted language:No compilation so no waiting to compile.
- Ability to execute code outside the program.
- A very well made using.
- MATLAB apps allow you to perform common engineering tasks without having to program. Visualize how different algorithms work with your data, and iterate until you've got the results you want.

3.2.2 Test instances

We consider several types of randomly generated instances for experimental analysis. These instances are generated in such a way that they reflect special properties. In almost all of these instances the weights (sizes) are uniformly distributed in a given interval with data range [1,R] where R is chosen suitably and profits (costs) are expressed as a function of the weights. The special construction of these instances depends on the function used to compute the profits and this actually defines the special properties associated with each groups of instances as described in the book by Kellerer[**1**] . We executed the algorithms on different types of knapsack instances.

Uncorrelated instances

Profits and weights of the items are selected randomly in the range [1,R]. There is no correlation between the weight values and the profit values. There can be a large gap between the profit and weight values and hence such instances are generally easy to solve. These type of instances reflect the situation where the profit does not depend on the weight

Weakly correlated instances

Weight of an item j, w_j is selected randomly in the range [1,R]. But unlike the uncorrelated instances the profit of item j, p_j is selected randomly in the range [w_j-R/10,p_j+R/10] such that,$p_j \leq 1$ Although the name suggests that, the relation between profit and weight of an item should not be very strong but actually the weakly correlated instances have a very high correlation between profit and weight of an item and they differ by only a few percent. This actually represents a scenario which is realistic in some sense, i.e. the return of an investment is actually proportional to the invested amount having some small tolerance of variation.

Strongly correlated instances

Unlike the uncorrelated instances there is a strong relation between the profit and weight of an item. The weights are distributed in [1,R], and profit of item j is defined as w_j+R/10. This type of instances correspond to real life situation in which the return is proportional to investment with some fixed charge for each investment.

Subset sum instances

In these type of instances,w_j weight are randomly distributed in [1,R] and $p_j = w_j$. These type of instances reflect the situation where the profit of each item is equal to the weight or proportional to the weight as in the subset sum problem.

3.3 Experimental results

Specifications of the machine on which we ran all the experiments are,

- Manufacturer: Lenovo.
- Model: Lenovo G500.
- Processor: Intel (R) Core (TM) i3-3110M CPU @ 2.40GHz 2.40GHz.
- Memory(RAM): 4.0 GB

In the tables showing the times of performance of the algorithms, and how it consummate the memory of computer as (megabyte).

3.3.1 Dynamic programming

In previous chapter we have the algorithm of the Dynamic programming method in knapsack problem , we can test it on an example. We coded the algorithm of this method by MATLAB .

Setup :

In our experiments we generated an uncorrelated and weakly correlated and strongly correlated and Sum subset instances we set R=1000,10000 and with n = 100, 1000, 10000, 100000, 200000,260000 items. The knapsack capacity is c = 1000, and $c = 0.5 * \sum_{j=1}^{n} w_j$, and we run this procedure and generate the result in the following data.

Results :

	Uncorrelated		Weakly Correlated		Strongly Correlated		Subset-Sum	
	R		R		R		R	
N	1000	10000	1000	10000	1000	10000	1000	10000
100	0.035 s	0.054 s	0.0308 s	0.0320 s	0.0314 s	0.0327 s	0.02987 s	0.0445 s
	584 MB	586 MB	586 MB	586 MB	584 MB	584 MB	587 MB	586 MB
1000	0.079 s	0.087 s	0.106 s	0.0725 s	0.075 s	0.076 s	0.0749 s	0.077 s
	590 MB	590 MB	592 MB	593 MB	591 MB	592 MB	594 MB	594 MB
10000	0.524 s	0.534 s	0.4999 s	0.5028 s	0.501 s	0.519 s	0.5252 s	0.703 s
	659 MB	660 MB	661 MB	661 MB	661 MB	661 MB	662 MB	662 MB
100000	5.277 s	5.057 s	4.843 s	4.8765 s	4.798 s	5.733 s	4.8536 s	5.574 s
	1350 MB	1354 MB	1352 MB	1353 MB	1352 MB	1302 MB	1353 MB	1353 MB
200000	9.788 s	7.929 s	10.098 s	10.519 s	10.128 s	10.206 s	10.314 s	11.2256 s
	2096 MB	2096 MB	2094 MB	2101 MB	2170 MB	2169 MB	2120 MB	2169 MB

TABLE 3.1 – The results on c=1000

	Uncorrelated		Weakly Correlated		Strongly Correlated		Subset-Sum	
	R		R		R		R	
N	1000	10000	1000	10000	1000	10000	1000	10000
100	0.037 s	0.040 s	0.045 s	0.039 s	0.0359 s	0.047 s	0.036 s	0.041 s
	556 MB	556 MB	559 MB	559 MB	555 MB	555 MB	559 MB	558 MB
1000	0.082 s	0.066s	0.075 s	0.078 s	0.10138 s	0.081 s	0.096 s	0.1189 s
	568 MB	567 MB	566 MB	566 MB	563 MB	563 MB	566 MB	566 MB
10000	0.636 s	0.417 s	0.525 s	0.521 s	0.514 s	0,524 s	0.726 s	0.768 s
	642 MB	636 MB	635 MB	635 MB	632 MB	631 MB	635 MB	635 MB
100000	5.064 s	3.682 s	4.799 s	4.982 s	4.852 s	4.927 s	7.039 s	7.106 s
	1327 MB	1326 MB	1326 MB	1326 MB	1323 MB	1323 MB	1326 MB	1326 MB
250000	13.036 s	9.958 s	14.518 s	14.5653 s	12.842 s	13.403 s	18.638 s	18.615 s
	2473 MB	2473 MB	2477 MB	2471 MB	2474 MB	2474 MB	2478 MB	2478 MB
260000	13.131 s	9.655 s	13.726 s	12.663 s	12.933 s	13.051 s	18.45 s	19.612 s
	2550 MB	2550 MB	2549 MB	2549 MB	2551 MB	2551 MB	2555 MB	2555 MB
267881	13.558 s	10.53 s	13.578 s	13.168 s	13.236 s	13.540 s	19.974 s	20.208 s
	2616 MB	2616 MB	2610 MB	2610 MB	2612 MB	2612 MB	2635 MB	2615 MB

TABLE 3.2 – The results on c $= 0.5 * \sum_{j=1}^{n} w_j$

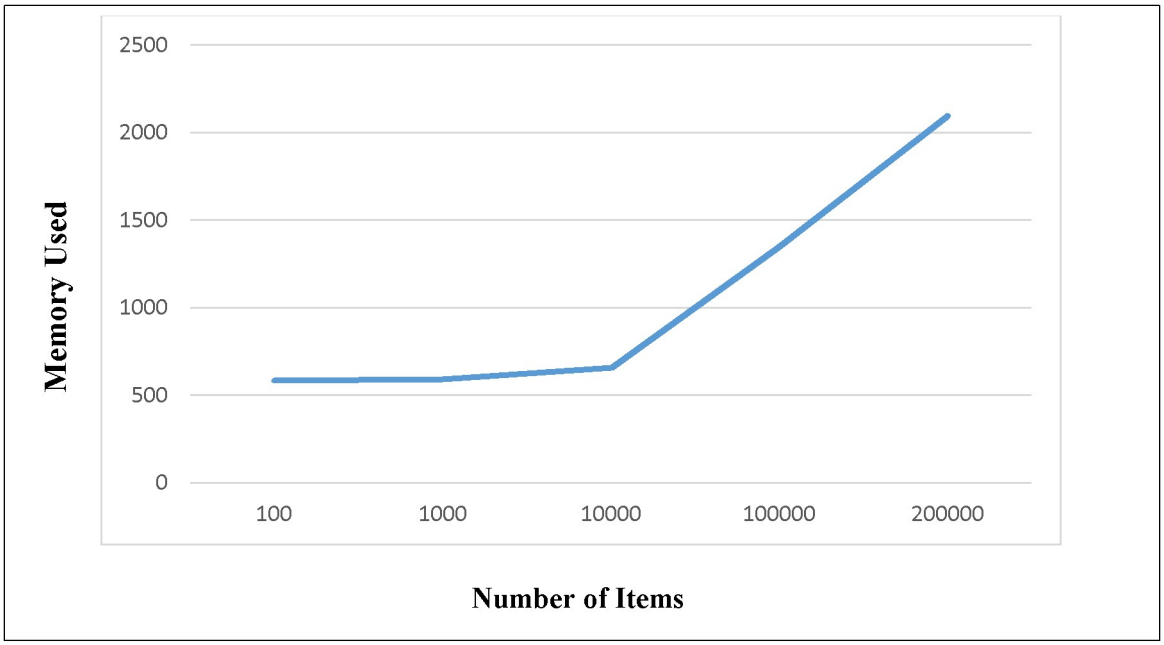

FIGURE 3.1 – The relation between number of items and memory used

we remark in the two previous table the method of dynamic programming is effective for finding the solution of knapsack problem and it very quickly for the calcul, but as we increase the number of items, the number of basic operations for the dynamic programming increase Since, dynamic programming require large memory for calcul. knowing that the memory of my computer was stopped in n= 260000.
The complexity of dynamic programming depends on the number of items and the capacity. Therefore, if we increase the capacity of the knapsack the number of basic operations and the memory required for the dynamic programming will increase. See(Figure 3.1).

3.3.2 Simulated annealing

In previous chapter we explained the application simulated annealing method in knapsack problem and how create the neighbourhood search of this problem. Then we coded the algorithm simulated annealing method by MATLAB . and we will run the program in several times.
Setup :

We generated an uncorrelated instance with n = 100, 500, 1000, 2000, 30000 items. The knapsack capacity is c = 500, 1000, weigth value is chosen randomly in [1,50], profits value is chosen randomly in [1,100], the initial temperature is 1000 and final temperature is 0.03 and we chose $\alpha \in [0, 1]$. finally we run this procedure and generate the result in data.

Results :

	max weight	alpha	weight	best profi	time of execusion /seconds
n=100	500	0,7	486	1658	0.09
		0,8	497	2072	0.12
		0,9	497	2204	0.22
n=500	500	0,7	486	1669	2.49
		0,8	474	1846	3.84
		0,9	498	2130	8.07
n=1000	500	0,7	487	2170	23.59
		0,8	479	1612	35.37
		0,9	500	1878	74.99
n=2000	500	0,7	493	1582	237.63
		0,8	488	2134	357.59
		0,9	486	1763	751.02
n=3000	500	0,7	483	2236	750.71
		0,8	494	1598	1220.33
		0,9	493	1771	2690.74

TABLE 3.3 – The results on c = 500

	max weight	alpha	weight	best profi	time of execusion /seconds
n=100	1000	0,7	817	2639	0.19
		0,8	980	2917	0.19
		0,9	995	2969	0.33
n=500	1000	0,7	846	2974	2.64
		0,8	935	4478	3.98
		0,9	976	3847	8.25
n=1000	1000	0,7	794	3050	21.88
		0,8	980	3833	34.43
		0,9	973	3583	79,06
n=2000	1000	0,7	888	3070	243.06
		0,8	988	3746	388.23
		0,9	972	3784	805.13
n=3000	1000	0,7	897	3010	751.60
		0,8	994	3992	1195.92
		0,9	968	4097	2899.73

TABLE 3.4 – The results on with c = 1000

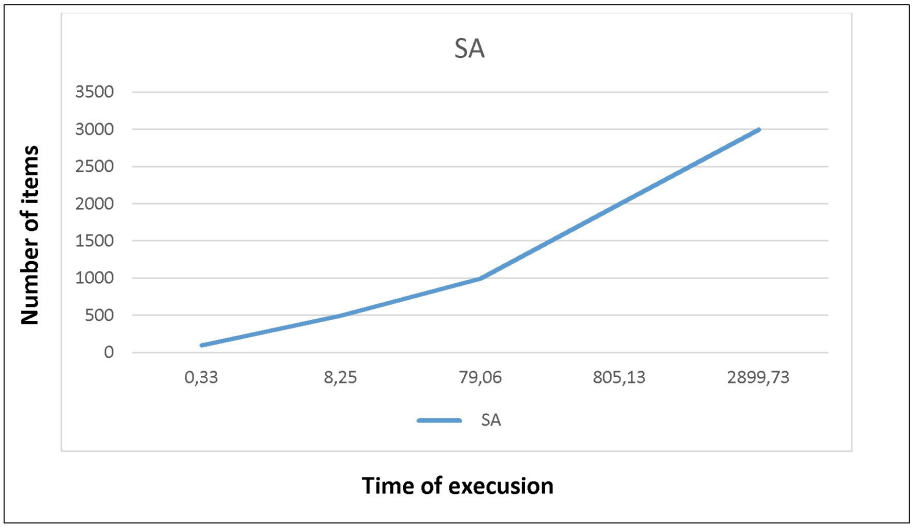

FIGURE 3.2 – The relation between the number of items and time of execusion

According to both tables, we notice that as α approches 1 the method give a good bound (solution), but infortunatly the execution time increase too.

3.4 Analysis of results

For the testing of the different algorithms, we generated files with different sizes where each record consists of a pair of randomly generated integers representing the weight and value of each item. We performed two types of testing. For the first one, we were increasing the number of items to be considered for the knapsack, while holding the capacity of the knapsack constant (equal to 500). During the second testing, we were increasing the capacity of the knapsack, while fixing the number of items to 5000.

3.4.1 Testing I: Increase the number of items and capacity = 500

- **10 items**

	10 Items				
	Total Weight	Max Value	Items included	Memory used	Operations
Simulated annaling	335	628	3, 6, 8, 9, 10	571 MB	630
Dynamic programming	335	628	3, 4, 6, 7, 8, 9	570 MB	5010
Greedy algorithm	335	628	3, 6, 8, 9, 10	572 MB	10
Brute force	335	628	3, 6, 8, 9, 10	572 MB	12286

TABLE 3.5 – Testing I : The results on 10 items

- **25 items**

	25 Items				
	Total Value	Max Value	Items included	Memory used	Operations
Greedy algorithm	484	1246	4, 6, 7, 9, 13, 19, 20, 21	564 MB	25
Brute force	497	1246	6, 7, 9, 13, 17, 19, 20, 21	564 MB	905969662

TABLE 3.6 – Testing I : The results on 25 items

The maximum number of items we could run the brute force algorithm for was 25. Moreover, Table 3.5 and Table 3.6 show us that so far dynamic programming and Simulated Annealing produce results that are the same as the optimal solution generated by the brute force algorithm.
Next we consider the solutions greedy, dynamic programming and Simulated Annealing generate in terms of the total value, average number of basic operations and memory used and time of execution.

- **100 items**

	100 Items				
	Total Weight	Total Value	Time (second)	Memory used	Operations
Simulated annealing	470	1555	0.305	570 MB	11699
Dynamic programmin	500	1556	0.021	570 MB	50100
Greedy algorithm	497	1379	0.008	568 MB	100

TABLE 3.7 – The results on 100 items

- **300 items**

	300 Items				
	Total Weight	Total Value	Time (second)	Memory used	Operations
Simulated annealing	468	1793	2.268	571 MB	35199
Dynamic programmin	500	2225	0.033	571 MB	150300
Greedy algorithm	500	1784	0.017	571 MB	300

TABLE 3.8 – The results on 300 items

- **500 items**

	500 Items				
	Total Weight	Total Value	Time (second)	Memory used	Operations
Simulated annealing	485	1790	8.287	573 MB	58099
Dynamic programmin	500	1795	0.036	573 MB	250500
Greedy algorithm	500	1787	0.024	573 MB	300

TABLE 3.9 – The results on 500 items

- **750 items**

	750 Items				
	Total Weight	**Total Value**	**Time (second)**	**Memory used**	**Operations**
Simulated annealing	**477**	**1701**	**27.23**	**573 MB**	**87099**
Dynamic programmin	**500**	**2138**	**0.036**	**574 MB**	**375000**
Greedy algorithm	**500**	**1745**	**0.035**	**574 MB**	**750**

TABLE 3.10 – The results on 750 items

- **1000 items**

	1000 Items				
	Total Weight	**Total Value**	**Time (second)**	**Memory used**	**Operations**
Simulated annealing	**500**	**2185**	**76.516**	**578 MB**	**119099**
Dynamic programmin	**500**	**2187**	**0.049**	**575 MB**	**501000**
Greedy algorithm	**500**	**2065**	**0.045**	**574 MB**	**1000**

TABLE 3.11 – The results on 1000 items

Moreover, we can conclude that the dynamic programming and Simulated Annealing, algorithms outperform the greedy algorithm in terms of the total value it generates.
We decided to further analyze the dynamic programming, Simulated Annealing and Greedy algorithms in terms of the number of basic operations (Fig. 3.3) for the different input files (number of items).

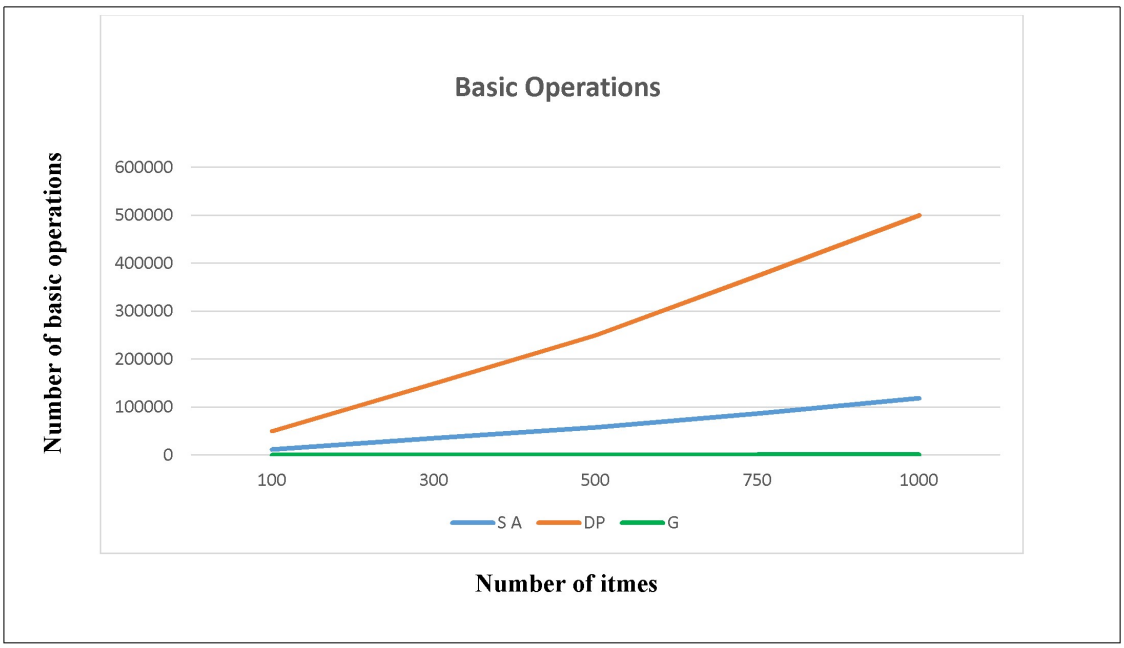

FIGURE 3.3 – Basic operations

3.4.2 Testing II: Increase the capacity and number of items = 500

- **Capacity = 100**

	C=100		
	Total Weight	**Total Value**	**Operations**
Simulated annealing	**97**	**2282**	**50500**
Dynamic programmin	**100**	**2337**	**50500**

TABLE 3.12 – The results on capacity = 100

- **Capacity = 200**

	C=200		
	Total Weight	**Total Value**	**Operations**
Simulated annealing	**198**	**3145**	**53599**
Dynamic programmin	**200**	**3399**	**100500**

TABLE 3.13 – The results on capacity = 200

- **Capacity = 300**

	C=300		
	Total Weight	**Total Value**	**Operations**
Simulated annealing	**299**	**3916**	**56099**
Dynamic programmin	**300**	**4021**	**150500**

TABLE 3.14 – The results on capacity = 300

- **Capacity = 400**

	C=400		
	Total Weight	**Total Value**	**Operations**
Simulated annealing	**398**	**4575**	**58599**
Dynamic programmin	**400**	**4675**	**200500**

TABLE 3.15 – The results on capacity = 400

- **Capacity = 500**

	C=500		
	Total Weight	**Total Value**	**Operations**
Simulated annealing	**496**	**5138**	**60599**
Dynamic programmin	**500**	**5238**	**250500**

TABLE 3.16 – The results on capacity = 500

We plot the results of the above tables in Fig. 3.4 .

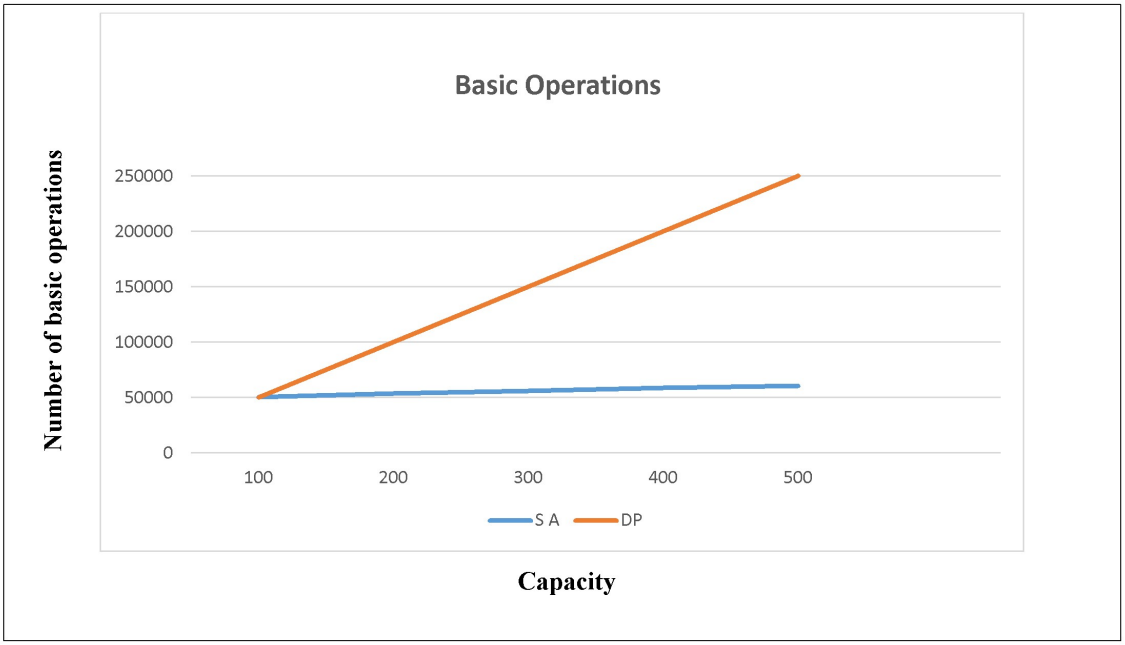

FIGURE 3.4 – Basic operations

As long as the capacity increase we notice that the dynamic programming number of operations and memory required will be a lot greater than the simulated annealing ones.

3.5 Conclusion

The comparative study of the brute force, greedy, dynamic programming and simulated annealing shows that while the complexities of these algorithms are known, the nature of the problem they are applied to makes some of them more suitable than others. The best approximation approaches for the 0/1 Knapsack Problem are dynamic programming and simulated annealing. As we have shown, the choice between the two depends on the capacity of the knapsack and the size of the population. However, one may decide to choose dynamic programming over simulated annealing in any circumstances, because it is easy and straightforward to code. In contrast, simulated annealing require a lot more time in terms of understanding the concepts of the paradigm.

Conclusion

The knapsack problem is a combinatorial optimization problem where one has to maximize the benefit of objects in a knapsack without exceeding its capacity. This problem although it contains a one constraint but it was classified as an NP-hard problem, Since, the researchers became interested in this problem. The problem has modeled into many industrial situations, such as cargo loading and capital budgeting, logistics like loading of airplanes or boats, the economy such as financial. management.

In this dessertation we explained the difference approach for solving knapsack problem and how to calculate the upper bound and the lower bound and we gave some examples for each method.
The knapsack problem can be solved by exact methods as branch and bound and dynamic programming that is guaranteed the optimality of solution, but they are very intensive in terms of computing time and memory requirements. For this reason we can use approximate methods that is reasonable and close to exact solution.

Our objective is apply methods for solving the knapsack problem, we applied the algorithms that it were described in chapter 2 for solving knapsack problem. We used the program MATLAB for programming our algorithms and we tested on four different data that can be uncorrelated, weakly correlated, or strongly correlated and sum subset instances. Lastly, we analyze the computational results.

The comparative study of the brute force, greedy, dynamic programming and simulated annealing shows that while the complexities of these algorithms are known, the nature of the problem they are applied to makes some of them more suitable than others. The best approximation approaches for the 0/1 Knapsack Problem are dynamic programming and simulated annealing. As we have shown, the choice between the two depends on the capacity of the knapsack and the size of the population. However, one may decide to choose dynamic programming over simulated annealing in any circumstances, because it is easy and straightforward to code. In contrast, simulated annealing require a lot more time in terms of understanding the concepts of the paradigm.

This dessertation is very much of exploratory nature. As such, there are countless related ideas and topics that one could study in more detail in the future. In the future, we will try to improve these algorithms to obtaining better solution in less time and in less memory consumption.

Bibliography

[1] Kellerer.H ,Pferschy.U, and Pisinger.D. "Knapsack problems". Springer, (2004).

[2] Papadimitriou.C.H, Steiglitz,.K. "Combinatorial optimization - algorithms and complexity". Prentice Hall, (1982).

[3] https://www.cs.cornell.edu/Courses/cs3110/2012sp/lectures/lec19-asymp/review.html.Date of access [17.04.2016 at 21:33:06].

[4] Garey.M.R, Johnson.D. "Computers and intractability: A guide to the theory of NP-completeness". New York: W.H. Freeman and Company.(1979).

[5] Bellman.R. "Dynamic Programming". Princeton University Press. (1957).

[6] Bellman.R. Dreyfus.S. "Applied Dynamic Programming". Princeton University Press. Princeton,NJ. (1962).

[7] Crainic, T.G., Toulouse, M.: "Parallel strategies for metaheuristics". In: Glover.F.W, Kochenberger.G.A, Handbook of Metaheuristics, pp. 475–513. Springer (2003)

[8] Feo.T.A, Resende.M.G.C. "Greedy randomized adaptive search procedures". J. Global Optim.6, 109–133 (1995)

[9] Feo, T.A , Resende, M.G.C. "A probabilistic heuristic for a computationally difficult set covering problem". Oper. Res. Lett. 8, 67–71 (1989).

[10] Glover.F. "Future paths for integer programming and links to artificial intelligence". Comput. Oper. Res. 13, 533–549 (1986).

[11] Gendreau.M, Potvin.J.Y, Chapter 6: Tabu search. In Burke, E.K., Kendall, G. (eds.) Search Methodologies, pp. 165–186. Springer (2006).

[12] Glover.F. "Parametric combinations of local job shop rules". In ONR Research Memorandum, No. 117, GSIA, Carnegie Mellon University, Pittsburgh (1963).

[13] Hedar.A, Ali.A. "Tabu search with multi-level neighborhood structures for high dimensional".

[14] Chelouah.R, Siarry.P," Tabu search applied to global optimization", Eur.J, Oper. Res. 123,256–270 (2000).

[15] Kirkpatrick.S, Gelatt.C., VecchiM. "Optimization by simulated annealing". Science 220, 671–680 (1983).

[16] Flynn.M. "Some computer organizations and their effectiveness". IEEE Trans. Comput. C-21, 948–960 (1972)

[17] Obitko, Marek, "Basic Description". IV. Genetic Algorithm. Czech Technical,(1998).

[18] Mitchell, Melanie. "An Introduction to Genetic Algorithms". Massachusettss The MIT Press, (1998).

[19] Freville.A. "The multidimensional 0-1 knapsack problem : an overview". European Journal of Operational Research, vol. 155, pages 1-21. (2004).

[20] Mathews.G.B. On the partition of numbers. Proceedings ofthe London Mathematical Society, 28,486-490, 1897.

[21] Martello.S, Toth.P. "Solution of multiple knapsack problem", European Journal of Operational Research, (1980)

[22] M. Fischetti, P. Toth, "A new Dominance Procedure for Combinatorial Optimization Problems", Operations Research Letters 7, (1988).

[23] Gallo.G, HammerP.L, and B. Simeone. "Quadratic knapsack problems". Mathematical Programming Study, 12:132-149. (1980).

[24] Witzgall.C. "Mathematical methods of site selection for electronic message systems". Technical report, Applied Mathematics Division, National Bureau of Standards, (1975).

[25] Rhys.J. "A selection problem of shared fixed costs and network flows". Management Science, 17.200-207, (1970).

[26] Johnson.E.L, Mehrotra.A, and Nemhauser.G.L., Min-cut clustering. Mathematical Programming, 62.133-152, 1993.

[27] Helmberg.C, Rendl.F, and Weismantel.R. "Quadratic knapsack relaxations using cutting planes and semidefinite programming". In Integer Programming and Combinatorial Optimization, 5th IPCO conference, volume 1084 of Lecture Notes in Computer Science, pages 175-189. Springer, (1996).

[28] Bellman.R. "Some applications of the theory of dynamic programming, a review", Operations Research, vol. 2, pp. 275-288, (1954).

[29] Ahrens.J.H., Finke.G. "Merging and sorting applied to the zero-one knapsack problem". Operations Research, Vol. 23, pp.1099–1109. (1975).

[30] Horowitz.E and Sahni.S, "Computing partitions with applications to the knapsack problem", Journal of ACM, vol. 21, pp. 277-292, (1974).

[31] Dantzig.G.B. "Discrete variable extremum problems". Operations Research, 5:266-277, 1957.

[32] Wolsey.L.A. "Integer Programming". Wiley.J. (1998).

[33] Martello.S and Toth.P. "A new algorithm for the 0-1 knapsack problem". Management Science, 34.633-644. (1988).

[34] Fayard.G and Plateau.G. "An algorithm for the solution of the 0-1 knapsack problem". Computing, 28:269-287. (1982).

[35] H. Miiller-Merbach. "Improved upper bound for the zero-one knapsack problem", A note on the paper by Martello and Toth. European Journal of Operational Research, 2.212-213,(1979).

[36] Dudzinski.K and Walukiewicz.S. "Exact methods for the knapsack problem and its generalizations". European Journal of Operational Research, 28:3-21. (1987).

[37] Pisinger.D. "An expanding-core algorithm for the exact 0-1 knapsack problems". European Journal of Operational Research. Vol. 87, pp:175–187.(1995).

[38] Ingargiola.G.P. and Korsh.IF, "Reduction algorithm for zero-one single knapsack problems". Management Science, 20:460-463. (1973).