THE DEMOCRATIC PEOPLE'S REPUBLIC OF ALGERIA
MINISTRY OF HIGHER EDUCATION AND SCIENTIFIC RESEARCH

UNIVERSITY MOHAMED BOUDIAF - M'SILA

FACULTY: Mathematics and Computer Science

DEPARTEMENT: Computer Science

N° :……..

Discipline: Mathematics and Computer Science

Branch: Computer Science

Specialty: Tools and Methods for Business Intelligence

Dissertation Presented to obtain
Master's Degree Academic

By: BOUCHAREB Sara

SUBJECT

Sieve Optimization Method

A Survey and Applications

Publicly supported before the jury composed of:

Pr.Gasmi Abdelkader	University of M'sila	President
Pr.Bouderah Brahim	University of M'sila	Supervised
Dr. Hemmak Allaoua	University of M'sila	Co-Superviser
Dr.Louanas Bilal	University of M'sila	Examiner

Promotion : 2016 /2017

To my Dear Parents,
All my Family and my Friends.

Acknowledgments

First and above all, I praise God, the almighty for providing me this opportunity and granting me the capability to proceed successfully.

I am deeply indebted to my supervisor Pr. BOUDERAH Brahim and Dr.HEMMAK Allaoua for their continuous supervision, wise advice, and support.

I greatly benefit from their guidance during this work. My sincere thanks go to all the people who have contributed in this search, by information, advice, criticism or encouragement.

This work appears in its current form due to the assistance and guidance of several people. I would therefore like to take this opportunity to convey my sincere thanks to all of them

Contents

List of Figures

List of Tables

General Introduction

Combinatorial optimization problems arise, in many forms, in various aspects of everyday life. Nowadays, optimization algorithms, enabling us to make the best use of the available resources while guaranteeing a level of service, drive many services. Examples of such services are public transportation, foods delivery, university timetabling, and patient scheduling.

The field of metaheuristics, artificial intelligence, and operations research, have been tackling many of these problems for years, without much interaction. However, in the last few years, such communities have started looking at each other's advancements, in order to develop optimization techniques that are faster, more robust, and easier to maintain.

Combinatorial optimization is a branch of optimization. Its domain is optimization problems where the set of feasible solutions is discrete or can be reduced to a discrete one, and the goal is to find the best possible solution. To deal with problems of combinatorial optimization, the objective is to find the best solution or optimal solution, one that minimizes a given cost function. There are some techniques to solve not complex problems, such as Branch and Bound or Branch and Cut. As the search space complexity grows up, the cost of those algorithms can increase exponentially, making the search of a solution not feasible. Another way to tackle these problems is to find a suboptimal solution but in a reasonable time. In some cases, we may even find the optimal solution to the problem. Such techniques can divided into two main groups' heuristics and metaheuristics.

The knapsack problem is believed to be one of the NP-hard problems. The knapsack problem or rucksack problem is a problem in combinatorial optimization is one of the widely and extensively studied resource allocation problem. In its basic version, we are given a number of items from which we are required to select a subset to carry it in a fixed capacity knapsack. Items differ by their value and their required place in the knapsack. The aim is to load items, which maximize the overall reward without exceeding the capacity.

Objective and Methodology

The aim of this dissertation is to analyze a new method, which is metaheuristic approache, and show how these can be used to solve NP-hard combinatorial optimization problems. Besides, what is the major advantages of using such techniques .however, is to propose a modified version of Sieve algorithm proposed by Pr.Bouderah, Dr.Hemmak [11] in the hope to enhance the quality of produced solutions for knapsack problem.

Layout

This work is divide into four chapters as follow

The first chapter consists on an introductory chapter of Reviews of combinatorial optimization, including the complexity class of problems (Class P, class NP and Class NP-complete), methods of resolution such as exact methods, approximate methods such as heuristics and metaheuristics.

The second chapter describe the State of art, which is a survey about the knapsack problems.

The third chapter, Sieve algorithm and explain the difference between this algorithm and the other ones have the same name. Explains the principal basics of sieve algorithm that have been applied to 0/1 Knapsack problem.

The fourth chapter Reports the results of an experimental performance evaluation of this approach on several well-known benchmark datasets and the last chapter concludes we compare with previous results.

Finally, **General conclusion and perspectives** of future research Ideas.

Chapter 1
The Combinatorial Optimization

Contents

1.1 Introduction

Combinatorial Optimization is a very interesting and challenging field of study. In the last 60 years, many works have been published in which several theoretical and practical optimization problems have been addressed with various approaches. Exact techniques have been applied in order to find optimal solutions to the given problems. Because of the complexity of the great part of the combinatorial problems, exacts are not able to prove optimality for "big-size" instances (where the concept of big size is strictly dependent from the nature of the problems addressed). Moreover, the theory of complexity shows us how it is unlike that a polynomial time algorithm will ever be developed for NP-Hard problems.

In order to provide good feasible solutions for those cases in which exacts fail to reach the optimality, or simply for improving the performances of the exacts, heuristic techniques have been developed. They consist in approximation approaches, obtained by applying a tailored strategy to the input problem, according to some given criteria, and leading to a feasible output solution. They are generally fast, but their behavior is strictly dependent from the instance addressed and can change consistently according to changes in the input.

However, combinatorial optimization is the study of optimization problems on discrete and combinatorial objects. This area includes many and important problems like travel salesman, knapsack, shortest paths. Combinatorial optimization find its applications in real life problems such as resource allocation and network optimization. In a broader sense, combinatorial optimization also has applications in many fields like artificial intelligence, machine learning and computer vision. The study of combinatorial optimization gives a unified framework to characterize the optimal value through min-max formula and to design efficient algorithm to compute an optimal solution. Given broad applications of combinatorial optimization, we are interested in designing fast algorithms to solve these problems.

Optimization problems divided into two major category, as exact and approximate, exact algorithms gives exact solution to the problem as name suggested. Approximate algorithm may give exact or may not give exact solution, in other words they gives approximate solutions to the problems. Approximate algorithm further divided into two major categories as heuristic and metaheuristics algorithms. Heuristics algorithm includes Local search, Divide and conquer, Branch-and-bound, Dynamic programming, cut & plane etc. metaheuristics algorithms includes evolutionary algorithm, genetic algorithm, scatter search, simulated annealing, tabu search, guided local search, hill climbing, Iterated local search, stochastic algorithm

1.2 Solution process of optimization problems

To solve optimization problems have been of major interest in operations research (OR) [4]. Planning is viewed as a systematic, rational, and theory-guided process to analyze and solve planning and optimization problems. The planning process consists of several steps: [5]

1) Recognizing the problem,
2) Defining the problem,
3) Constructing a model for the problem,
4) Solving the model,
5) Validating the obtained solutions,
6) Implementing one solution.

1.3 Problem Instances

We have seen in the previous section how the construction of a model is embedded in the solution process. When building a model, we can represent different decision alternatives using a vector $x = (x_1, \dots, x_n)$ of n decision variables. We denote an assignment of specific values to x as a solution. All solutions together form a set X of solutions, where $\boldsymbol{x} \in X$.

Decision variables can be either continuous ($\boldsymbol{x} \in \mathbb{R}^n$) or discrete ($\boldsymbol{x} \in \mathbb{Z}^n$). Consequently, optimization models are either continuous where all decision variables are real numbers, combinatorial where the decision variables are from a finite, discrete set, or mixed where some decision variables are real and some are discrete. Typical sets of solutions used for combinatorial optimization models are integers, permutations, sets, or graphs.

We want to distinguish between problems and *problem instances*. An instance of a problem is a pair (X, f), where X is a set of feasible solutions $x \in X$ and $f : X \to \mathbb{R}$ is an evaluation function that assigns a real value to every element x of the search space. A solution is feasible if it satisfies all constraints. The problem is to find an $x^* \in X$ for which:

$f(x^*) \geq f(x)\ for\ all\ x \in X$ (Maximization problem) (1.1)

$f(x^*) \leq f(x)\ for\ all\ x \in X$ (Minimization problem), (1.2)

x^* Is called a globally optimal solution (or optimal solution if no confusion can occur) to the given problem instance.

An *optimization problem* is defined as a set I of instances of a problem. A problem instance is a concrete realization of an optimization problem and an optimization problem can be viewed as a collection of problem instances with the same properties and which are generated in a similar way. Most users of optimization methods are usually dealing with problem instances, as they want to have a better solution for a particular problem instance. Users can obtain a solution for a problem instance since all parameters are usually available. Therefore, it is also possible to compare the quality of different solutions for problem instances by evaluating them using the evaluation function f .

(1.1) and (1.2) are examples of definitions of optimization problems. However, it is expensive to list all possible $x \in X$ and to define the evaluation function f separately for each x. A more elegant way is to use standardized model formulations. A representative formulation for optimization models that is understood by people working with optimization problems as well as computer software is: 1.3 Search Spaces

$$\begin{aligned} \text{minimize} \quad & z = f(x), \\ & \text{subject to} \\ & g_i(x) \geq 0, \qquad i \in \{1,\ldots,m\}, \\ & h_i(x) = 0, \qquad i \in \{1,\ldots,p\}, \\ x \in W_1 \times W_2 \times \ldots \times W_n, \quad & W_i \in \{\mathbb{R},\mathbb{Z},\mathbb{B}\}, \qquad i \in \{1,\ldots,n\}, \end{aligned} \tag{1.3}$$

Where *x* is a vector of *n* decision variables $x1,\ldots,xn, f\ (x)$ is the objective function that is used to evaluate different solutions, and $g(x)$ and $h(x)$ are inequality and equality constraints on the variables xi. $\mathcal{B}$ Indicates the set of binary values {0,1}.

In standard optimization problems, there are decision alternatives, restrictions on the decision alternatives, and an evaluation function. Generally, the decision alternatives are modeled as a vector of variables. These variables are used to construct the restrictions and the objective criteria as a mathematical function. By formulating them, we get a mathematical model relating the variables, constraints, and objective function. Solving this model yields the values of the decision variables that optimize (maximize or minimize) values of the objective function while satisfying all constraints. The resulting solution is referred to as an optimal feasible solution.

1.4 Search Spaces

In optimization models, a search space X is implicitly defined by the definition of the decision variables x ∈ X. We have different important aspects of search spaces. Metrics that can be used for measuring similarities between solutions in metric search spaces. Neighborhoods in a search space are defined based on the metric used. Finally, in combinatorial search spaces where a metric is defined, we can introduce the concept of fitness landscape. All this aspects of search spaces to find locally and globally optimal solutions.

1.5 Properties Of Optimization Problems

The purpose of optimization algorithms is to find high-quality solutions for a problem. If possible, they should identify either optimal solutions x^*, near-optimal solutions $x \in X$, where $f(x) - f(x^*)$ is small or at least locally optimal solutions.

Problem difficulty describes how difficult it is to find an optimal solution for a specific problem or problem instance. Problem difficulty is defined independently of the optimization method used. Determining the difficulty of a problem is often a difficult task, as we have to prove that there are no optimization methods that can better solve the problem. Statements about the difficulty of a problem are method independent, as they must hold for all possible optimization methods.

Any optimization problem have properties starts with complexity classes, which allow us to formulate bounds on the performance of algorithmic methods. This allows us to make statements about the difficulty of a problem. Then, we continue with properties of optimization problems that can be exploited by modern heuristics. Besides, the locality of a problem and presents corresponding measurements.

The locality of a problem is exploited by guided search methods, which perform well if problem locality is high. Important for high locality is a proper definition of a metric on the search space. Finally, the decomposability of a problem and recombination-based optimization methods exploit a problem's decomposability.

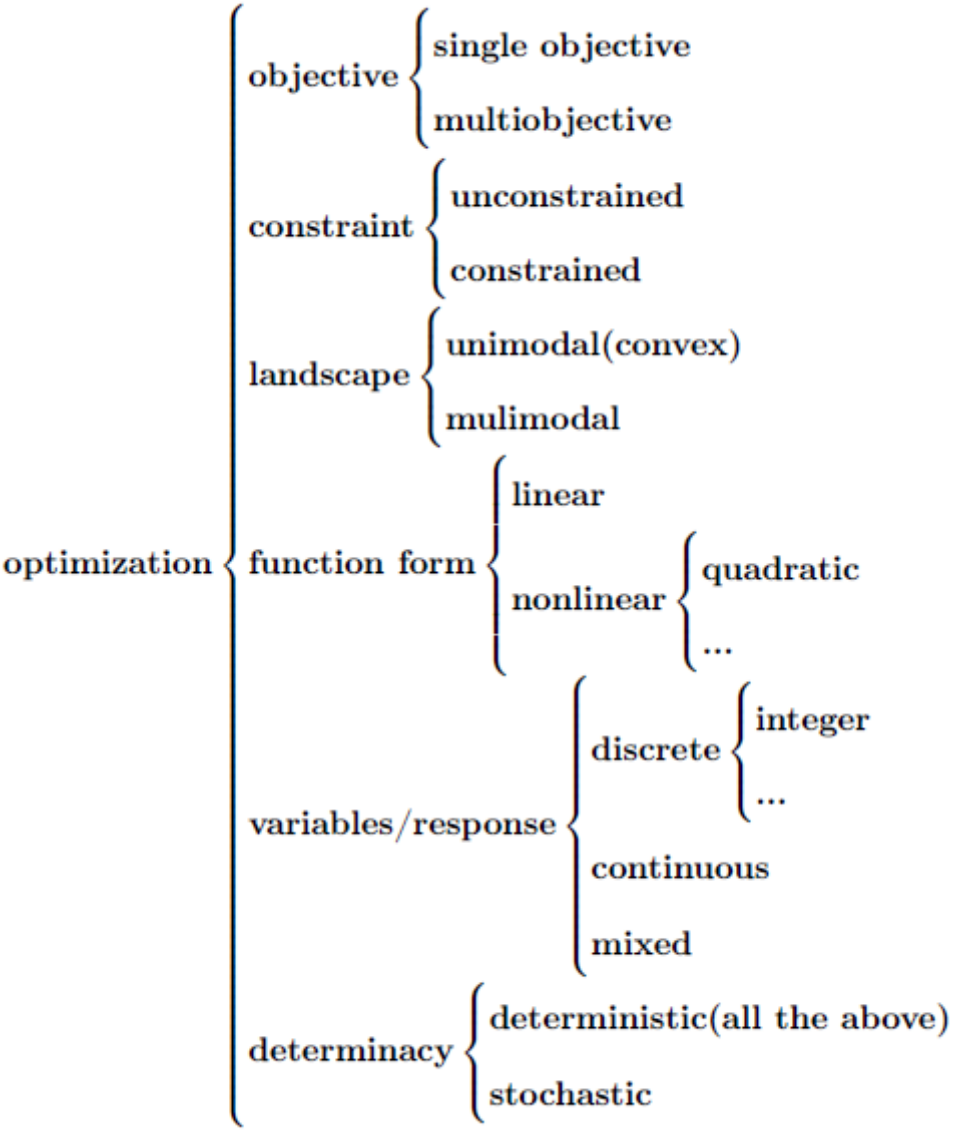


Figure 1.1: Classification of optimization problems. [18]

1.6 The Different complexity Classes

Before discussing the different methods of solving combinatorial optimization problems, we introduce some definitions and notions of complexity (POC).

Generally, the execution time is the major factor that determines the effectiveness of an algorithm, and then the time complexity of an algorithm is the number of instructions (assignment, comparison, algebraic operations, reading and writing, etc. . .) that includes the algorithm for solving any problems.

1.6.1 Class P

Class P consists of all those problems that can be solved on a deterministic Turing machine in polynomial time from the size of the input. Turing machines are an abstraction that used to formalize the notion of algorithm and computational complexity. A comprehensive description of them can be found in [9]

1.6.2 Class NP

NP is a complexity class that represents the set of all decision problems for which the instances where the answer is "yes" have proofs that can be verified in polynomial time. The NP stands for non-dermenistic time; NP class is that the set of all decision problems that can be solved by polynomial time non-deterministic algorithms, where non-dertminitec means that no particular rule is follows to make the guess. This means that if someone gives us an instance of the problem and a certificate (sometimes called a witness) to the answer being yes, we can check that it is correct in polynomial time.

1.6.3 Class NP- Complete

Among all the problems belonging to NP, a subset contains the most difficult problems called NP complete problems. An NP-complete problem has priority than any problem in NP transformed (reduced) by one in polynomial time. This means that a problem is NP-complete when all problems belonging to NP reducible to it.

If there is a polynomial algorithm for an NP-complete problem, it automatically finds a polynomial resolution of all problems of class NP. [9]

1.6.4 Class $NP - Hard$

The class NP-hard problems are partly similar but more difficult problems than NP-complete problems. They do not themselves belong to class NP (or if they do, nobody has invented it, yet), but all problems in class NP can be reduced to them. Very often, the NP-hard problems really require exponential time or even worse. Notice that NP-complete problems are a subset of NP-hard problems, and that is why NP-complete problems are sometimes called NP-hard. It can also happen that some problem, which is nowadays known to be only, NP-hard will be proved NP-complete in the future – it is enough that somebody just invents a nondeterministic Turing machine, which solves the problem in polynomial time.[13]

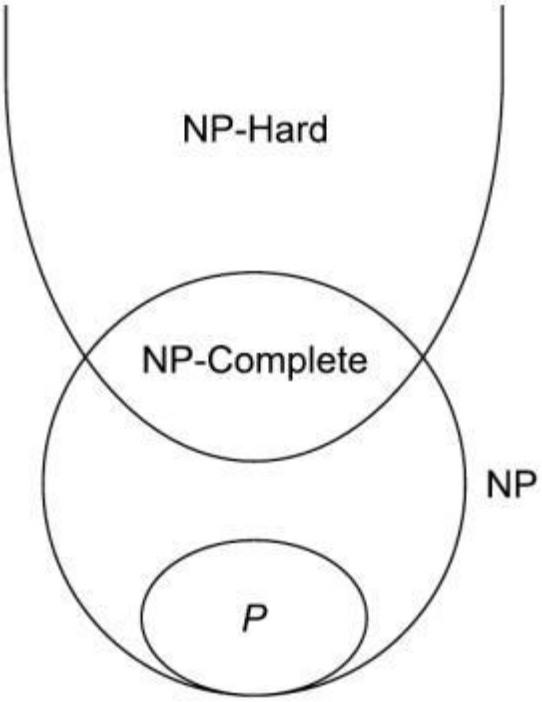


Figure 1.2: The relationships among complexity classes of problem [18]

1.7 Solving Combinatorial Optimization Problems

Two classes of algorithms are available for the solution of combinatorial optimization problems: exact and approximate algorithms.

1.7.1 Exact Methods

Exact algorithms are guaranteed to find the optimal solution and to prove its optimality for every finite size instance of a combinatorial optimization problem within an instance dependent run time. In the case of NP-hard problems, exact algorithms need, in the worst case, exponential time to find the optimum. For most NP-hard problems the performance of exact algorithms is not satisfy factory.

1.7.1.1 Dynamic programming

Dynamic programming is a method that in general solves optimization problems that involve making a sequence of decisions by determining, for each decision, sub problems that can be solved in like fashion, such that an optimal solution of the original problem can be found

from optimal solutions of sub problems. This method based on Bellman's Principle of Optimality. [2]

1.7.1.2 Branch and Bound

Branch and bound B&B is an algorithm design paradigm for discrete and combinatorial optimization problems, as well as mathematical optimization. A branch-and-bound algorithm consists of a systematic enumeration of candidate solutions by means of state space search: the set of candidate solutions is thought of as forming a rooted tree with the full set at the root. The algorithm explores *branches* of this tree, which represent subsets of the solution set. Before enumerating the candidate solutions of a branch, the branch is checked against upper and lower estimated *bounds* on the optimal solution, and is discarded if it cannot produce a better solution than the best one found so far by the algorithm. [24]

1.7.2 Approximate Methods

1.7.2.1 Local Search

Local search is a general approach for finding high-quality solutions to hard combinatorial optimization problems in reasonable time. It is based on the iterative exploration of neighborhoods of solutions trying to improve the current solution by local changes. The types of local changes that may be applied to a solution are defined by a neighborhood structure.

Definition 1.1 A neighborhood structure is a function $N: S \mapsto 2^S$ that assigns a set of neighbors $N(s) \subseteq S$ to every $s \in S$. $N(s)$ is also called the neighborhood of s.[4]

The choice of an appropriate neighborhood structure is crucial for the performance of a local search algorithm and is problem-specific. The neighborhood structure defines the set of solutions that can be reached from s in one single step of a local search algorithm. Typically, a neighborhood structure is defined implicitly by defining the possible local changes that may be applied to a solution, and not by explicitly enumerating the set of all possible neighbors.

The solution found by a local search algorithm may only be guaranteed to be optimal with respect to local changes and, in general, will not be a globally optimal solution.

Definition 1.2 A local optimum for a minimization problem (a local minimum) is a solution s such that $\forall\, s' \in N(s)$: $f(s) \leq f(s')$. Similarly, a local optimum for a maximization problem (a local maximum) is a solution s such that: $\forall\, s' \in N(s)$: $f(s) \geq f(s')$.[4]

A local search algorithm also requires the definition of a neighborhood examination scheme that determines how the neighborhood is searched and which neighbor solutions are accepted. While the neighborhood can be searched in many different ways, in the great majority of cases the acceptance rule is either the best-improvement rule, which chooses the neighbor solution giving the largest improvement of the objective function, or the first-improvement rule, which accepts the first improved solution found.

1.7.2.2 Simulated Annealing

Simulated annealing is a metaheuristic devised by Kirkpatrick *et al.* (1983) and derives its name and approach from the behavior of metals and glass as they systematically heated and re-heated and then allowed to cool steadily. SA inspired by an analogy between the physical annealing of solids (crystals) and combinatorial optimization problems. The objective, as with other metaheuristics, is to obtain a close approximation to the global optimum for a given problem. [10]

In the physical annealing process a solid is first melted and then cooled very slowly, spending a long time at low temperatures, to obtain a perfect lattice structure corresponding to a minimum energy state. SA transfers this process to local search algorithms for combinatorial optimization problems. It does so by associating the set of solutions of the problem attacked with the states of the physical system, the objective function with the physical energy of the solid, and the optimal solutions with the minimum energy states.

SA is a local search strategy, which tries to avoid local minima by accepting worse solutions with some probability. In particular, SA starts from some initial solution
s and then proceeds as follows: At each step, a solution $s' \in N(s)$ is generated (often this is done randomly according to a uniform distribution).If s' improves on s it is accepted; if s' is worse than s,then s' is accepted with a probability which depends on the difference in objective function value $f(s) - f(s')$, and on a parameter T,Called temperature .T is lowered (as is also done in the physical annealing process) during the run of the algorithm, reducing in this way

the probability of accepting solutions worse than the current one. The probability p_{accept} to accept a solution s' is often defined according to Metropolis distribution [10]

$$p_{accept}(s, s', T) = \begin{cases} 1, & if\ f(s') < f(s); \\ \exp\left(\dfrac{f(s) - f(s')}{T}\right), & otherwise. \end{cases}$$

1.7.2.3 Tabu Search

Tabu search (TS) based on the premise that problem solving, in order to qualify as intelligent, must incorporate adaptive memory and responsive exploration. The adaptive memory feature of TS allows the implementation of procedures that are capable of searching the solution space economically and effectively. Since local choices are guided by information collected during the search, TS contrasts with memoryless designs that heavily rely on semi-random processes that implement a form of sampling. The emphasis on responsive exploration (and hence purpose) in tabu search, whether in a deterministic or probabilistic implementation, derives from the supposition that a bad strategic choice can often yield more information than a good random choice. [14]

1.7.2.4 Ant Colony Optimization

The Ant Colony Optimization metaheuristic (ACO) is a recent technique used for solving combinatorial optimization problems. The source from which ACO takes inspiration is the use by ants of pheromones as a communication method (laying it on the ground as a guide for the next ants). In analogy to the biological example, ACO is based on the indirect communication of a colony of simple agents, called artificial ants, mediated by artificial pheromone trials.

The pheromone trial in ACO is a distributed numerical information, which is used by the ants for probabilistically constructing solutions to the problem being solved and updated, by the same ants, during the execution of the process. In spite of many cases in which ACO could not reach the results obtained by other metaheuristic techniques, the approach is still being used to address several optimization problems, among which quadratic assignment, vehicle routing, sequential ordering and scheduling. [6][7][15]

The behavior of real ants seem to be able to find their way (from the nest to a food source and back, or around an obstacle) with relative ease, although they are almost blind [17].

Ethological studies discovered that this capacity is the result of the interplay via chemical communication between ants (through a substance called pheromone) and an emergent phenomenon caused by the simultaneous presence of many ants. We define an ant to be an agent with the following properties:

- The ant remembers already visited towns using an ordered list called tabu list (TL) for this purpose (caution, TL here is different from that of Tabu Search),
- At every step the ant chooses, using a probabilistic rule, a town to move to among those not in the TL,
- After a tour has been completed the ant lays a trail τ_{ij}on each arc $(i;\ j)$ used (trail is the analog of pheromone) and clears its tabu list.

1.7.2.5 Genetic Algorithm

Genetic algorithms (GA) are population based stochastic search algorithms inspired by Darwin's Theory of Evolution and were first introduced by John Holland in the early 70's. Ata high level, the GA begins with a randomly generated population of potential solutions. Each member of the population is represented by a DNA string, a chromosome, that is an encoding of the problem and search space. Members of the population are selected for recombination based on fitness scores and recombined to form a new population, i.e., the next generation. This process is repeated until some termination criteria are satisfied, e.g., number of generations, run time, or improvement of the best solution. [12]

The GA's population based approach gives it implicit parallel computational abilities and also allows for program level parallelism. GAs can be classified by how they create the next generation, being either generational or steady-state. With a generational based GA, there are two separate populations: parents and children. The parent population is used to select members from for breeding, placing the resulting offspring into the child population.

The generational GA delays the offspring's reproductive participation until the next generation. The main benefit of this type of GA is it is less likely to prematurely converge on a possibly inferior solution. The downside is that good individuals, those with high fitness scores, have to wait until the next generation before their genetic material is used. In short, this methodology delays the offspring's reproductive participation. A steady state GA maintains one population where the offspring are placed back into the population with, or replacing, their

parents. Because of this single population implementation, in steady-state GAs, the offspring's reproductive participation is immediate, but it is also more likely to prematurely converge on a solution than its generational counterpart.[17]

The simplest form of GA involves three types of operators:[12]

- **Selection:** This operator selects chromosomes in the population for reproduction. The fitter the chromosome, the more times it is likely to be selected to reproduce.
- **Crossover:** This operator exchanges subsequences of two chromosomes to create two offspring. For example, the strings

  10000100 and 11111111

  Could be crossed over after the third locus in each to produce the two offspring

  10011111 and 11100100

This operator roughly mimics biological recombination between two single-chromosome (haploid) organisms. (Higher organisms have chromosomes in pairs and are thus "diploid.").

- **Mutation:** This operator randomly flips some bits in a chromosome. For example, the string 00000100 might be mutated in its second position to yield 01000100. Mutation can occur at each bit position in a string with some probability, usually very small (e.g., 0.001).

Algorithm 1 Genetic Algorithm

1: $t \leftarrow 0$
2: initialize $P(t)$
3: evaluate $P(t)$
4: **while** termination conditions not met **do**
5: Recombine $P(t)$ to yield $C(t)$ via *crossover* and *mutation* // Reproduction
6: evaluate $C(t)$
7: select $P(t+1)$ from $P(t)$ and $P(t)$ // Selection
8: $t \leftarrow t+1$

[18]

1.7.2.6 Particle swarm optimization

Particle swarm optimization (PSO) is a population based stochastic optimization technique developed by Dr. Eberhart and Dr. Kennedy in 1995, inspired by social behavior of bird flocking or fish schooling.

PSO shares many similarities with evolutionary computation techniques such as Genetic Algorithms (GA). The system is initialized with a population of random solutions and searches for optima by updating generations. However, unlike GA, PSO has no evolution operators such as crossover and mutation. In PSO, the potential solutions, called particles, fly through the problem space by following the current optimum particles.

Each particle keeps track of its coordinates in the problem space which are associated with the best solution (fitness) it has achieved so far. (The fitness value is also stored.) This value is called *pbest*. Another "best" value that is tracked by the particle swarm optimizer is the best value, obtained so far by any particle in the neighbors of the particle. This location is called *lbest*. when a particle takes all the population as its topological neighbors, the best value is a global best and is called *gbest*.

The particle swarm optimization concept consists of, at each time step, changing the velocity of (accelerating) each particle toward its p*best* and *lbest* locations (local version of PSO). Acceleration is weighted by a random term, with separate random numbers being generated for acceleration toward *pbest* and *lbest* locations.

In past several years, PSO has been successfully applied in many research and application areas. It is demonstrated that PSO gets better results in a faster, cheaper way compared with other methods.

Another reason that PSO is attractive is that there are few parameters to adjust. One version, with slight variations, works well in a wide variety of applications. Particle swarm optimization has been used for approaches that can be used across a wide range of applications, as well as for specific applications focused on a specific requirement.[22]

1.7.2.7 Bee colony optimization

In the period 1999-2003, Dusan Teodorovic (adviser) introduced the basic concepts of BCO under the name Bee System, and Panta Lucic (Ph.D. candidate) while doing research at Virginia Tech. BCO is a nature inspired metaheuristic method developed for efficiently finding solutions to difficult combinatorial optimization problems. [16]

The basic idea behind BCO is to build the multi agent system (colony of artificial bees) that will search for good solutions of various combinatorial optimization problems, exploring the principles used by honeybees during nectar collection process. Artificial bee colony usually consists of a small number of individuals, but BCO principles are gathered from the natural systems. Artificial bees investigate through the search space looking for the feasible solutions. In order to find the best possible solutions, autonomous artificial bees collaborate and exchange information. Using collective knowledge and information sharing, artificial bees concentrate on the more promising areas and slowly abandon solutions from the less promising ones. Systematically, artificial bees collectively generate and or improve their solutions. The BCO search is running in iterations until some predefined stopping criterion is satisfied. [1]

1.8 Conclusion

We have described the domain of combinatorial optimization, which is where our work stands. We have the origin of the implicit complexity of the optimization problems that belong to this class. We have mention relevant methods of combinatorial optimization problem

Chapter 2
Knapsack Problems

Contents

2.1 Introduction

The knapsack problem (KP) is a well-known and widely studied combinatorial optimization problem. Researchers are interested in the study of KPs due to its theoretical and practical importance. Theoretically, it has a simple structure and the ability to solve more complex optimization problems. In addition, practically, the KP models a wide range of industrial situations belonging to the domains of transportation as cargo loading, cutting stock, telecommunication, reliability, advertisement, budget allocation, financial management and cryptography.[19]

The KP derives its name from the hiker's problem of selecting which items, among a given set of items, to fill his knapsack in such a way that the overall value of selected items is maximized such that the knapsack weight capacity is not exceeded.

KP has seen several variations over the years; the difference lies in the items and resources distribution, the objectives considered, etc. In its binary form, an item is either taken or left while in the fractional form, the decision maker can load only a fraction of the item. In the bounded KP, there are a number of copies from each item type, and an unlimited number of copies in the unbounded form. In the case of the multiple-choice KP, group while in the multidimensional classes items and the multiple forms, a number of knapsacks are supposed to be filled. If the weight and the value of each item vary depending on the knapsack it is assigned to, the problem is called multiobjective 0-1 KP.[19]

Knapsack problems are among the most intensively studied NP-hard combinatorial optimization problems. The applications of these problems span a wide canvas from industrial applications and financial management to electronic commerce and personal health-care. The common flavour in most of these problems is resource allocation. The allocation of a specific amount of a single resource among competitive alternatives is often modelled as a knapsack problem or its variants.[28]

2.2 Static knapsack problems

The static aspect of KPs is defined as given a collection of items, each with a certain value and weight, and a limited capacity knapsack, which items should be packed in the knapsack in order to maximize contents value. The common characteristics of static KPs are:[19]

- Items available at the same time
- Items values and weights are known beforehand

However, other objectives and specifications were considered, giving rise to the appearance of several variants of the problem. We focus in the rest of this section on the most important.

We begin by explaining the binary form of the KP. Then we consider a generalization; the bounded KP, followed by a special case; the subset sum problem. After that, the multiobjective KP, an extension of the binary form, as well as the minimization version. Finally, we present the stochastic variant. Table 1.1 illustrates resolution methods and applications of studied variants.

2.2.1 The 0-1 Knapsack Problem (0-1 KP)

The binary KP is the basic form of KPs. The 0-1 KP is defined as: given a set of n items, with different sizes and values, and a knapsack with a finite capacity c, the objective is to select a collection of items, which maximize the total value without exceeding the knapsack capacity.[8]

$$\text{Maximize} \quad Z(x) = \sum_{i=1}^{n} v_i x_i \qquad (2.1)$$

$$\text{Subject to} \quad \sum_{i=1}^{n} w_i x_i \leq c,$$

$$x_i = 0 \text{ or } 1; \qquad i = 1, 2, \ldots, n$$

A Knapsack Problem holds when we are asked to select a number of items, from a given set, to be carried in a knapsack. Each item is associated with value vi and weight wi, and the knapsack has a limited capacity c. The KP is to fill the knapsack in such a way that its content has maximum value under the capacity constraint.

2.2.2 The Bounded Knapsack Problem (BKP)

The BKP is a generalization of the KP01 where from each item type there are a single copy, $b_j = 1$ for all $j \epsilon n$. In the BKP, from each item type there are up to b_j items of type j that can be put in the knapsack. [3]

The problem is then stated as:

$$\begin{aligned} &\text{Maximize} \quad Z(x) = \sum_{i=1}^{n} v_i x_i \\ &\text{Subject to} \quad \sum_{i=1}^{n} w_i x_i \leq c, \\ &\qquad x_i \in \{0, 1, 2, ..., b_i\}, \quad i = 1, 2, ..., n \end{aligned} \tag{2.2}$$

The BKP is closely related to the KP01, that is why all mathematical and algorithmic approaches of the KP01 could be extended to solve the BKP, and BKPs could be converted to the binary model.

The bounded version of the KP can be extended to the unbounded KP (UKP), where an unlimited number of copies of each item type is available.

2.2.3 The Subset Sum Problem (SSP)

The SSP is a special case of the KP, arising when the weight of an item is equal to its reward. The objective is to select the subset of items, with the largest weight, which still fit in the knapsack. Formally, given n items, each with an associated weight w_j, and a knapsack with a finite capacity c, find the collection of items whose total weight combination is equal or closest to c. [19]

The problem can be stated as:

$$\begin{aligned} &\text{Maximize} \quad Z(x) = \sum_{i=1}^{n} w_i x_i \\ &\text{Subject to} \quad \sum_{i=1}^{n} w_i x_i \leq c, \\ &\qquad x_i = 0 \text{ or } 1; \qquad i = 1, 2, ..., n \end{aligned} \tag{2.3}$$

The SSP is known to be NP-hard and it can be solved in a pseudo-polynomial time. The SSP can be solved simply by any method of the 0-1 KP; this implies some specific treatments.

2.2.4 The Multi-Objective 0-1 Knapsack Problem (MOKP)

In the MOKP, a binary KP is considered with m knapsacks of capacities $c1;\ c2;\ \dots;\ cm$ (i.e., m objectives and m constraints). The profit and weight of each item are varying according to which knapsack the item is placed v_{ij} and w_{ij} are respectively the profit and the weight associated with assigning item i to knapsack j.[19]

We formulate the problem as:

$$\begin{aligned} &\text{Maximize} && Z(x) = (z_1(x), z_2(x), \dots, z_m(x)) \\ &\text{Where} && z_i(x) = \sum_{j=1}^{n} v_{ij} x_j \\ &\text{Subject to} && \sum_{j=1}^{n} w_{ij} x_j \le c_j, && j = 1, 2, \dots, m, \\ & && x_{ij} = 0 \text{ or } 1, && \text{for all } i \text{ and } j \end{aligned} \tag{2.4}$$

This problem arises often in the problem of assigning jobs to several machines, in loading problems, etc.

2.2.5 The Minimization Knapsack Problem (MinKP)

All maximization problems have equivalent minimization versions, which supposed to minimize the cost (instead of the profit in the maximization version) or to minimize the profit of the unloaded items. [19]

The MinKP is a transformed version of the 0-1 KP to a minimization problem. The objective is to minimize the total profit of the unselected items while their combined weight is at least equal to $y = \sum_{i=1}^{n} w_i - c$

$$\begin{aligned} &\text{Minimize} && Z(x) = \sum_{i=1}^{n} v_i y_i \\ &\text{Subject to} && \sum_{i=1}^{n} w_i y_i \ge y, \\ & && y_i = 0 \text{ or } 1; && i = 1, 2, \dots, n \end{aligned} \tag{2.5}$$

Table 2.1: Summary of Literature Review on Static KPs [19]

			SOLUTION APPROACHES				APPLICATIONS
			EXACT		HEURISTIC		
KNAPSACK PROBLEMS	STATIC PROBLEMS	*KP01*	DP	▶ Bellman (1950), Fayad & Plateau (1975), Pisinger (1995), Martello & Toth (1997)	PTAS	▶ Capara et al. (1997)	Cargo Loading
			B&B	▶ Kolesar (1967), Martello & Toth (1980), Pisinger (1997)	FPTAS	▶ Lawler (1979), Magazine & Oguz (1981)	Capital budgeting
			DP and B&B	▶ Martello & Toth (1984), Plateau & Elkihel (1985)			Cutting stock
		BKP	DP	▶ Pferschy (1999)	GREEDY	▶ Martello & Toth (1990)	
			B&B	▶ Tamir (2009)	PTAS	▶ Ingargiola & Korsh (1977), Pisinger (2000), Kellerer et al. (2004)	
		SSP	DP	▶ Martello & Toth (1984), Pisinger (1999), Samo & Toth (2002)	FPTAS	▶ Kellerer et al.(1997)	Two processor scheduling
					LOCAL SEARCH	▶ Ghosh & Chakravarti (1999)	Cargo Loading
					GA	▶ Wang (2003)	
		MOKP	DP	▶ Klamroth & Wiecek (2000), Bazgan et al. (2009)	MOEA	▶ Zitzler & Thiele (1999) (2001), Alves & Almeida (2007)	
			B&B	▶ Visée et al. (1998)	HYBRID MOEA	▶ Knowles & Core (2000)	Assigning jobs to several
			LABELING ALG.	▶ Captivo et al. (2003)	GA	▶ Ben Abdelaziz et al. (2000)	machines
					TABU SEARCH	▶ Ben Abdelaziz et al. (2000)	
		DCKP	B&B	▶ Yamada et al. (2002) ▶ Hifi and Michrafy (2006)	LAGRANGIAN LOCAL SEARCH and TABU LIST	▶ Yamada et al. (2002) ▶ Hifi and Michrafy (2006)	
			CONFLICT GRAPHS	▶ Pferschy & Schauer (2009)	FPTAS	▶ Pferschy & Schauer (2009)	
		SKP	DP	▶ Morton & Wood (1998)	MONTE CARLO	▶ Morton & Wood (1998)	
			B&B	▶ Kosuch & Lisser (2008)			
			GRADIENT METH.	▶ Kosuch & Lisser (2008)			
		MinKP			GREEDY	▶ Güntzer & Jungnickel (1999)	

2.4 Statics about knapsack problem

The knapsack problem has been studied for more than a century, with early works dating as far back as 1897. The name "knapsack problem" dates back to the early works of mathematician Tobias Dantzig (1884–1956), and refers to the commonplace problem of packing your most valuable or useful items without overloading your luggage.[25]

We tried to do some statics about the knapsack problem publications over the years like we see in figuer 2.1.

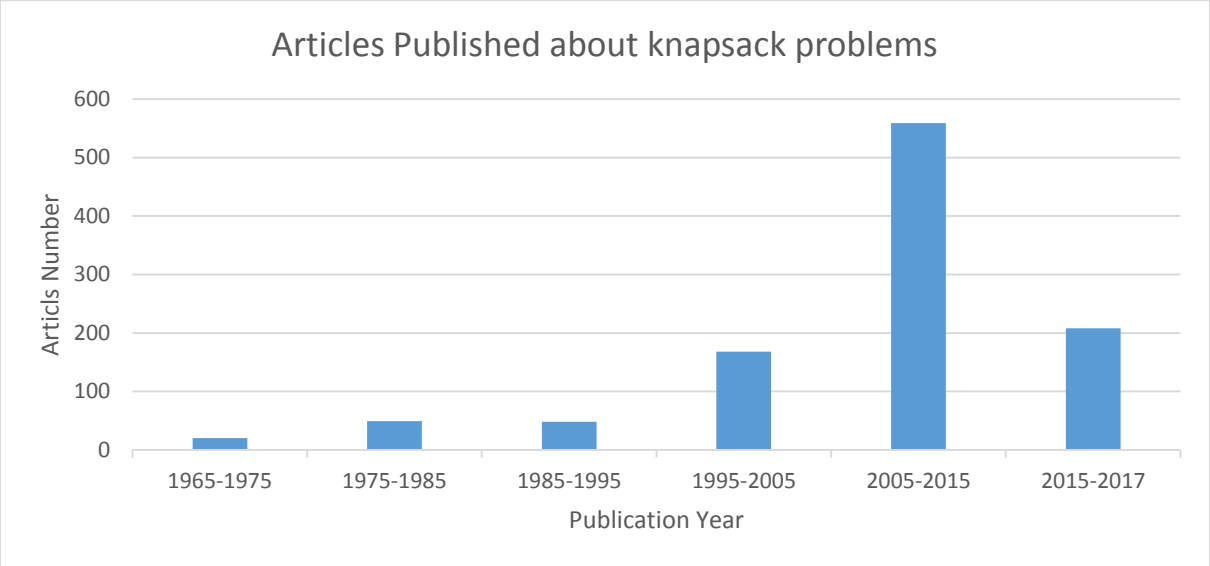


Figure 2.1 Articles Published about knapsack problems [23]

2.5 Conclusion

In this chapter, we reviewed the existing literature of the KP. We underlined the importance of the problem, which showed to be suitable to model several real world, after that we have collected some other types of knapsack problems and we have done some statics about this problem and its publication's over the years.

Chapter 3
The Sieve Method

Contents

3.1 Introduction

The purpose from this chapter is to understand the method we are going to use and to focus in its fundamental concepts in order to tackle the 0/1 knapsack problem.

This method, which is called “Sieve”, and presented, by Pr.Bouderah and Dr.Hemmak in a paper published in 2013, is considered as a metaheuristic method proposed to various combinatorial problems.

In single scheduling machine, the method provides very close solutions to optimality and is among effective, to tackle the difficult problems at hand.

Therefore, by using the proposed approach sieve method they propose mathematical model and formulation to this approach, they try to fill these gaps and give an alternative to treat combinatorial problems.

As other meta-heuristics such as genetic algorithms, ant colonies, and simulated annealing are inspired from natural phenomena, the proposed approach is also inspired from the grains sieve phenomenal where, in order to sift them, the grains are added by handfuls periodically into the sieve. For each handful, they are sift: it implies that the little grains fall through the holes of the sieve and the big grains move towards the center of the sieve by proportional speeds to their volumes. These two steps are repeated for a great number of times.

3.2 Preliminary

Before describing Sieve method for combinatorial optimization problems, let us begin by answering some preliminary questions, such as: what is a sieve, and what is sieve methods, beside that the definition of sieve algorithm proposed.

3.2.1 What is a sieve?

A device with meshes or perforations through which finer particles of a mixture (as of ashes, flour, or sand) of various sizes may be passed to separate them from coarser ones, through which the liquid may be drained from liquid-containing material, or through which soft materials may be forced for reduction to fine particles.[21]

3.2.2 What is sieve methods?

Sieving is the process of separating particles by size. The process started in ancient times in Egypt where sieves were used to separate grain. Today there are high tech methods such as laser diffraction and image analysis used to determine particle size distribution. Other methods such centrifuge techniques and sedimentation are also used, and there is another Sieve which is a language that can be used to create filters for electronic mail. Besides, the sieve of Eratosthenes which is an ancient (276-194 B.C.E) method used for primary numbers. However, sieves are the most commonly used devices for particle size analysis. The sieving process is comparatively inexpensive, simple in concept and easy to use. [20]

3.2.3 Definition of Sieve algorithm

Sieve algorithm it is new approach for solving combinatorial optimization problems as scheduling problems, traveling salesman problem, transport problems, and images segmentation which have exponential complexity and known as NP-hard problems. This approach, known as Sieve approach is based on the sieving operation idea used to sift grains by translating it on an algorithmic tool.

This approach generates randomly and iteratively a great number of feasible solutions by batches. The bad items are removed, while the good items are assembled in a smaller central set according to an appropriated fitness function. The best solution is computed from this small set according to the problem objective. The fitness function may be easy to compute but it may simulate the objective. Sieve method have a mathematical formulation for representing the problem environment.

3.3 Overview of Sieve Algorithm

In this section, we explain the main object of the proposed approach in details. It starts with the method principles, showing its parameters, and then, follows by the formulation of the general algorithm.

3.3.1 Sieve Principles

By analogy to the sieving operation, from which is inspired the approach, we have to satisfy following principles:

- The sieve is used to sift grains for getting the best ones.
- Grains to be sift are added into the sieve periodically by handfuls.
- The initial sieve is chosen according a sample of the grains to be sift.
- The grains can be divided on a great number of handfuls.
- For each added handful, the grains are sift many times.
- When sieving, the little grains fall out the sieve from holes and the biggest ones move slowly toward the sieve center; if all grains fall, we must change sieve by another one having smaller holes.
- The movements of grains are assumed linear and decelerated but their speeds are proportional to their volumes, bad items may never reach the center.
- If the operation is not well accomplished, we can also change the sieve by another one having larger holes.
- At least, the best grains will stabilize in a small circular area near the sieve center.

We should do good projection of these principles on the problem environment may provide good results. However, this projection will impose the intervention of several parameters, which could also affect the solution quality.

3.3.2 Sieve Parameters

Before using this method, we need to carefully define following such data of the problem, some parameters to be used by the principles of sieve method (figure. 3.1):

Parameters	Domain	Description
Sieve S		set of grains initially empty, it is the solutions pool
Radius r	$r \in R^+$	the radius r of the sieve taken large enough to hold a significant number of grains
Radius b	$b \in R^+$, $b \approx {}^{r}/_{10}, {}^{r}/_{20} \dots$	the radius b of the critical area A, b = small fraction of r, this critical area is used to accelerate the approach, so, no need to search the solution in S, but just in the area A;
Diameter h	$h \in R^+$ $h \ll r, h \approx lower\ bound$ $h\ increase\ h = \alpha * h\ ; \alpha > 1$ $h\ reduce\ h = \beta * h\ 0 < \beta < 1$	The diameter h of the hole sieve, h ≈ lower bound, h can be increased to changing sieve to improve results, a big value of h can produce divergence of the algorithm in which case we could reduce. So, the initial value of h must be chosen according the problem properties.
Quantity n grains	$n \in N$,$n \approx 100,200, \dots$	the quantity n of grains in each handful
Grains gi	$i = 1,2, \dots, n$	gi is a solution of the problem, well codified as vector, set, array, string, object, …
Distance di	$di \in R^+ di < r$, $i = 1,2, \dots, n\ of\ grains\ gi$ (1) $b < di < r$	the respective distances di, to the sieve center these distances will decrease during the movement of grains (when sieving): the grains approach to the center, initially we have (1) to avoid making bad grains in zone A;
fitness function f	$f(gi) \in R^+$	A fitness function f to evaluate grains, where $f(gi)$ = diameter of gi = function simulating the problem objective.
speed vi	$d_i^{t+1} = vi * d_i^t\ ; 0 \leq vi \leq 1$	The speed vi of grain movement, vi function of f.
Number Nb of iterations	$Nb \approx 1000,10000, \dots$	Nb is number of added handfuls, generally taken large enough.

Number N_s of sifts	$N_s \approx 10,20, \ldots$	N_s Number of sieving times for each added handful, also it is the number of grains movement.

The values of these parameters are chosen according to the problem treated. However, the change of these values can considerably improve the results.

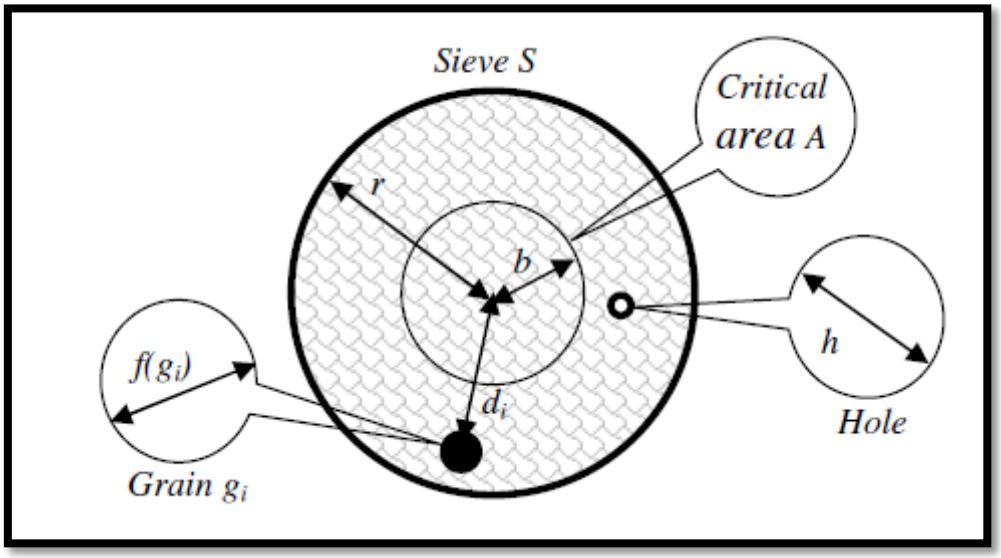


Figure 3.1: Sieve Parameters

3.3.3 General Sieve Algorithm

The general algorithm of sieve approach is composed of steps which are repeated for a great number of iterations, then consists just to get the best solution from a small central area (critical area A) of the sieve when the criteria stopping:

Sieve Algorithm

1: Choose initial Paramters;

2 : Initialize Sieve ;

3 : Evaluate_grains; ()

4 : ***Repeat***

5 : *Sieveing*();

6 : *Moving*();

7 : *Adding*();

8 : *Evaluate_grains*() ;

9 : *Update hole*;

10: ***Until***(***Critaria Stopping***);

Before starting sieving, we should initialize the parameters and sieve, which means select good holes diameter. In terms of combinatorial problems, this initial value of holes diameter will represent the lower (or upper) bound of problem solutions.

Therefore, it depends of the problem and it must be carefully chosen to accelerate the approach and avoid premature convergence of the algorithm.

While evolving in the approach, this value will be adjusted to the problem environment.

The first step consists to generate randomly a lot (handful) of grains (feasible solutions for the proposed problem). The size *n* of the handful, since the number of grains added at each time is depending of the problem. So, the grains gi $(i = 1,2, \dots, n)$ and their respective distances di from the center are generated randomly by using uniform law to avoid favoring certain solutions. [11]

The second step represents the sieving operation itself. It consists to move grains by applying the sieve principles, so:[11]

- When sieving, the grains g_i such as f(gi) $\leq$ h will fall out the sieve, and the grains gi such as f(gi) $>$ h will move slowly toward the sieve center with appropriated speed v_i;f(gi) must imperatively simulate the problem objective to be optimized; it may be easy to compute because it concerns a large set of feasible solutions (gi), but when computing the best solutions (in the area A), we will use the objective itself; that means:

the grains are moving according a fitness function f (their diameters), but the best on are chosen according their real objectives (their real weight); so some bad grains can arrive to the area A but they are not selected (false items); because f simulate the objective ($f \approx Obj$);

- v_i must be proportional to f(gi), that means: the grains which have a great diameter f(gi) will move more quickly toward the center and vice versa; that will favorite the best solutions to be selected and otherwise; because of congestion around the center, the speed v_i of grain must be a decreasing function of time: more grains approach the center, the choice is more rigorous;
- After Ns sieving iterations, if there are no improvement of the solutions, we can change the sieve, that means increasing the hole diameter by replacing h by $\alpha * h(\alpha > 1)$ as $\alpha = 1.1,1.2,1.3, \ldots$. this will bring down more grains having higher diameters, since that will increase the speed of remaining seeds. Longer solution candidates will enter the critical zone A;
- In the other hand, to avoid algorithm divergence, since when we obtain $S = \emptyset$, we reduce h by replacing it by $\beta * h$ $(0 < \beta < 1)$ as $\beta = 0.9,0.8,0.7, \ldots$;
- this second step will be repeated for Ns times of sieving, ;
- the two steps are also repeated Nb times of adding handfuls;

The third step is moving and evaluating the population of solutions and we take the best ones

The last step allows finding the best solution from the central critical area A which have radius b, that will accelerate the algorithm because we will just explore this small area instead of sieve entire contents;

3.4 The Modeling of Sieve Approach

Firstly, as in any other method, we have to carefully define a system codification of the sieve contents *S* and the solutions (grains) gi .This codification has a great importance and a significant influence on the approach efficiency. For example, gi can be integer vector, binary array, string of characters, set of items, and complex object.

Examples:

- in single machine scheduling problems, gi can be an integer sequence of jobs $(j_1, j_2, \ldots, j_k)$;
- in job shop scheduling problems, gi can be an integer matrix of jobs

$$j_{11}, j_{12}, \ldots, j_{1k},$$
$$j_{21}, j_{22}, \ldots, j_{2k},$$
$$\ldots \ldots \ldots \ldots \ldots \ldots \ldots$$

- in salesman travelling problems, gi can be an integer sequence of cities $(j1, j2, \dots, jk)$;
- in images segmentation problems, gi can be a binary array of digits 1 and 0;
- in search engine problems, gi can be a string of characters like URL web sites;
- in affectation problems, gi can be a set of couples as $\{(pi, sj)\}$; ....

Then, we must establish the rule of the fitness function f to be used to evaluate the grains gi by simulating the problem objective to be optimized and the rule of the speed $v_i(gi)$ of grain gi by using fitness function $f(gi)$.

Finally, we must choose suitable values for the parameters defined above $Nb, N_s, n, r, b, \alpha, \beta$.

Note that, in this approach, we have always to maximize f. Therefore, if we have to minimize the problem objective, we must adjust rules to satisfy these constraints, we can take for example: [11a]

$$f(g_i) = 1 - Objective'(g_i)/\sum_S Objective'(g_i)$$
$$v_i(g_i) = (1 - d_i/r)(1 - f(g_i)/\sum_S f(g_k)) \quad (3.1)$$

Where *objective'* is a sample function that simulating the objective, it must be easy to compute to accelerate the approach, however it must be proportional to the objective.

It is clear that all grains have the same function f but they have different speeds v_i. This speed v_i may be chosen such as bad items may move too slowly toward the center or they may never reach it. Bad items means gi such $f(gi) > h, i.e. f(gi) = h + \varepsilon \approx h$.

The first rule shows that $f(gi)$ is depending of all both grains gi, it means that the choice of a solution is depending of the others ones.

The second rule shows that $v_i(gi)$ is proportional against both $f(gi)$ and di. That means in one hand, the big grains will arrive quickly to the critical area before the little ones, in the other hand, when we increase h, some gi will fall, so the sum $\sum f(g_k)$ will decrease, it imply that $v_i(gi)$ will increase. That means, when some grains fall, that will allow the other grains to move more quickly.

Since we have initially, $b < di < r$, we will assume the set $S - A$ at the set S. It is clear that no need to sift or generate grains in zone A. In the other hand, the algorithm must not terminate by empty set A. So the number of iterations Nb must be enough great to avoid this bug.

In fact, we can take other rules for $f(gi)$ and $v_i(gi)$ but we must always respect the method principles to obtain good results.

3.5 Statement of the problem

We consider a set of items, from which we have to select a subset to be packed in a knapsack. The knapsack has a limited capacity. Items arrive randomly, over n discrete periods, without any prior data, and it joins the evaluation process once arrived. Each evaluation has to lead to an immediate decision, to accept the item or to leave it. Once an item is selected, the associated reward is received, the solution will be evaluated if it has a high probability we toke as best solution if it is not we leave and evaluate again. The bad items will be removed from population of solution to avoid bed solution; The translation of the above algorithm on a programming code produces the following program:

Sieve program:

S=∅ ; A=∅// Initialisations Parameters

$Initialization\ ()\ //initialize\ objective\ problem\ function$

$For\ (i = 1;\ i <= Nb;\ i++)$

Sieving ()

$\{\ v_i = 2;$

$if(Fitness(g_i) < h\ or\ v_i \leq v_{max})$

{

$For(j = 1; j <= n; j++)//add\ handful\ of\ n\ grains\ into\ sieve\ S$

{ $population(i,j) = 0$

$c = 0$

$generate\ gi//\ randomly\ by\ using\ uniform\ law$

$while\ \ c + pi(j) <= capacity$

{

$population(i,j) = 1$

$S = S \cup \{gi\}$

$Do\ generate\ gi$

$until\ population(i,j) = 0$

}

$v_i = v_i + c$

$population(i, Rnd()) = 1 - population(i, Rnd())$

$if\ Not\ Feasible\ \{population(i, Rnd()) = 1 - population(i, Rnd())\}$

}

}

} **//End Sieving**

$\boldsymbol{Moving}()$

$\{\ \ Probability\ = 0.9$

$For(i = 1; i <= n; i++)$

{

$generate\ gi$

$if(gi <= probability)$

{
$generate\ gi;$
}
}} //**End Moving ()**

$\boldsymbol{Evaluation}()$

{
$For(i = 1; i <= n; i++)$
{
$if\ (Objective(g_i) < min)$
{ $Objective(g_i) = min$
$solution = population(i, j)$ }}} **END**

To well complete this implementation, we must add the code for some functions:

- *Objective* : according the treated problem which is 0/1 knapsack problem ;

$maximaze \quad z = \sum_{i=1}^{n} p_i x_i \qquad (3.2)$

$\sum_{i=1}^{n} w_i x_i \leq C\ , \qquad (3.3)$

- *Fitness function* : $Fitness(i) = \frac{Objective(i)}{\sum_{i=1}^{n} Objective(i)}$
- vi: according fitness function f
- *Probability:* we take the probability of solution too high to avoid the bad grains in the critical area A

We develop the above general algorithm on a computer program to be tested on 0/1 knapsack problem. Since, this program could be immediately implemented once; we have carefully defined the necessary elements of the problem as the objective function and some parameters values. The main operations of this program are given below and follows by explanation.

- *Initializing variables;*
- *Generating handful;*
- *Sieving grains;*
- *Moving grains;*
- *Evaluating grains*
- *Getting best grain;*

Initializing variables

$S = \emptyset$ *: S* is a set of grains, the contents of the sieve, initially empty. At any step of the algorithm we have: $S = \{z = \sum_{i=1}^{n} w_i \ / \ z \leq capacity\}$; $A = \emptyset$ *: when sieving we will obtain* $A = \{y = \sum_{i=1}^{n} p_i \ / \ y = Max_{pi}\}$;

Generating handful

Randomly generate n grains and their respective distances to the center. In our case we generate items where cover all space feasible solutions.

Sieving grains

$If\ Fitness\ (i) < h\ Or\ v_i \leq v_{max}$ we take $S = S - \{g_i;\}$; $A = A \cup \{g_i;\}$ we store the best solution associated to the best fitness ,$v_i = v_i + c$: This operation allows us to change all grains positions by updating (decreasing) their distances to the center. Here, we consider that the movements of grains decrease normally by the constant "c" that we define in begin.

Moving grains

Moving the grains by define a probability $p = 0.9$ to take just the best solution, if the probability of solution less than the "p" we move the grains randomly again to avoid bed grains until we get the high probability solution.

Evaluating grains

We evaluate the grains by the fitness and objective function for each gi of A

For $fitness(gi) = Fitness(gi) / \sum_{i=1}^{n} Objective(gi)$

$$h = \sum_{i=1}^{n} Fitness(gi) /n$$

Getting best grains

$if\ (Objective(gi) < min)\ \{\ min = Objective(gi)\ ;\ solution = gi\ ;\}$

This operation consists to compute the best solution population found and its objective.

3.6 Conclusion

We proposed a solution approach for the 0/1 knapsack problem that incorporates a functions loading process in this case to be able to make decisions throughout the searching process, for each incoming item.

In this chapter, we define the sieve method and parameters with general algorithm, where this approach consists random and iterative batches of solutions in a great set. We ameliorate some mathematical formulation for this method to get best solution.

Chapter 4
Analyze Of Results

Contents

4.1 Introduction:

In this chapter, we will experiment the algorithm that we used in a standard dataset comparing its results with approach method results.

This chapter deals with the experimental investigation. Experimentation obtained by means of the proposed algorithm will be presented. We are looking to measure the effectiveness and the performance of our algorithm to produce a satisfiable solution. Therefore, we develop some metrics to show the effectiveness of our results.

4.2 Experimentation of results:

We have implemented this approach with VB language, and we have tested it under Windows 7 operating system with Intel(R) core™ i3-3217 CPU @ 430.

Sieve algorithm is built for different instance, and items sizes. For small n, we derive a solution strategy for knapsack problem. With large size items, generating a whole decision strategy is practically insignificant.

Therefore, we derive the optimal solution and near to optimal. The knapsack capacity is then fixed to 20%, 60%, and 80% of the sum of items weight.

The remainder of the present chapter is organized as follows. The second section will be concerned by the description of the experimentation's settings. We will differentiate thereafter between small and large instances. For each problem size, we present results of built instances and discuss the obtained values.

4.3 The Experimental settings

Experiment results for the proposed algorithm are presented in previous algorithm. We have considered small and large size items and instances . For small size items and instances, the algorithm generates a whole solution strategy considering the different manners to fill the knapsack with the objects. At each stage, we calculate the expected utilities of the present items with all possible ranks. With large size items and instances, the optimal solutions derive to the near optimal. Item features (value and weight) are positive integer variables.

4.4 Interpretation of the results

This section discusses the interpretation of the obtained results. We conduct two directions of experimentation: small size items and large size items for the sieve algorithm. Note that algorithm results were compared with Dynamic programming algorithm, where items are assumed present simultaneously to the evaluation.

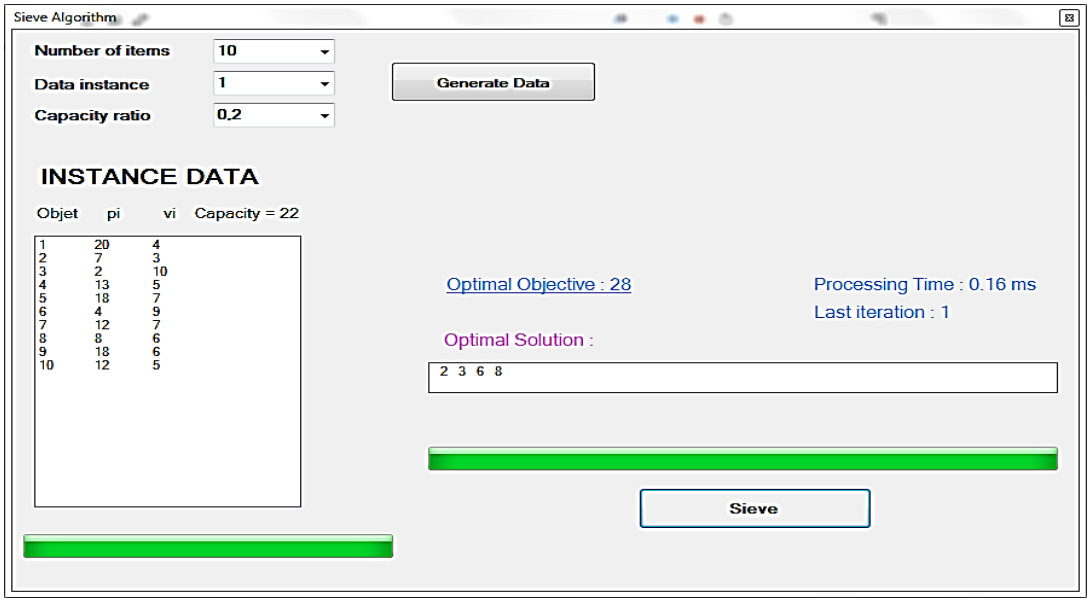


Figure 4.0 Example of sieve in screenshot

4.4.1 Small size items

We consider first small instances for which we derive solution strategies. Number of items were considered with n within the range {5, 40}, and number of instances with capacity ratio for each instance.

The overall results are presented in Table 4.1, where the minimum (Min), the average (Avg) and the maximum (Max) of capacity knapsack. Table columns refer respectively to the number of items (n), the number of instances (# instances), the optimal solution of dynamic programming, error, the computational time (CPU).

$$error = \frac{|S_{SM} - S_{DP}|}{S_{DP}}$$

We have calculate the value of sieve method in table 4.1

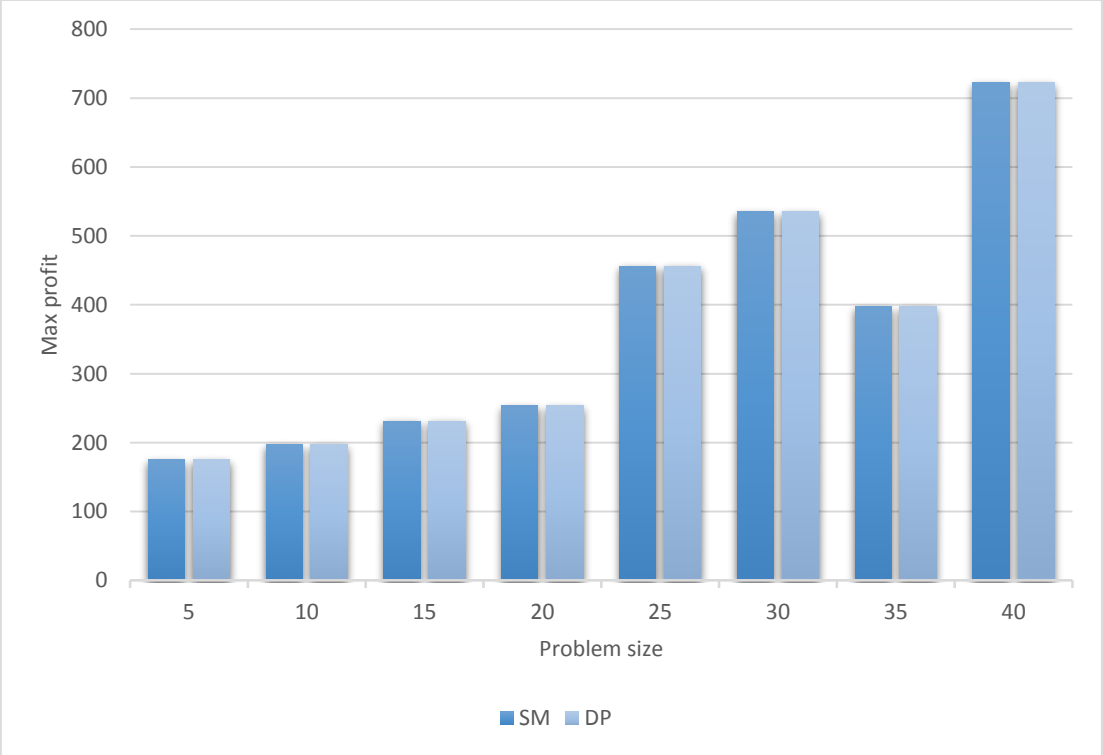
$$Value_{SM} = \frac{\sum_{i=1}^{10} instances}{10}$$

Table 4.1: Experimental results for the SM with small size items

n	*# instances*	CAPACITY RATIO	SM	DP	ERROR	CPU SM	CPU DP
		Min = 0.2	17.5	17.5	0	0,00001	0,004
5	10	Avr = 0.6	19.7	19.7	0	0,00001	0,004
		Max = 0.8	23.1	23.1	0	0,00001	0,003
		Min = 0.2	25.4	25.4	0	0,007	0,005
10	10	Avr = 0.6	45.6	45.6	0	0,007	0,004
		Max = 0.8	53.5	53.5	0	0,007	0,004
		Min = 0.2	39.7	39.7	0	0,027	0,006
15	10	Avr = 0.6	72.2	72.2	0	0,027	0,005
		Max = 0.8	81.5	815	0	0,027	0,006
		Min = 0.2	53.7	53.7	0	0,071	0,007
20	10	Avr = 0.6	93.6	93.6	0	0,071	0,008
		Max = 0.8	107	107	0	0,072	0,008
		Min = 0.2	65	65	0	0,154	0,01
25	10	Avr = 0.6	112.7	112.7	0	0,154	0,01
		Max = 0.8	130	130	0	0,156	0,01
		Min = 0.2	77.9	77.9	0	0,297	0,012
30	10	Avr = 0.6	137	137	0	0,224	0,012
		Max = 0.8	152.5	152.6	0,000655308	0,32	0,014
		Min = 0.2	91.5	91.5	0	0,512	0,014
35	10	Avr = 0.6	156.1	156.1	0	0,509	0,017
		Max = 0.8	172.8	173	0,0011560694	0,546	0,015
		Min = 0.2	107.3	107.3	0	0,864	0,015
40	10	Avr = 0.6	184.0	184.4	0,0021691974	0,846	0,016
		Max = 0.8	203.7	204.7	0,0048851979	0,865	0,016

Sieve method generates various possibility of loading. Each possibility represent a collection of different items. Evidently, the SM is growing with the problem size as shown in figure 4.1.

However, the SM in small size instances {5, 40} generates the optimal solutions from different valuation directions. Therefore, sieve algorithm supplied a variety of solutions in good time. Figure 4.1 is a comparison histogram of SM and the DP of the same instances obtained statically.



Figuer 4.1 Progress of the number of possible collections of items in terms of problem sizes with small items.

Our SM generates various possibility of small size items. Each possibility of size represent a collection of different items instances. Evidently, the SM is equivalent with the DP in small problem size as shown in figure 4.1.

The SM results are about 99.99 % of the optimal values when we compare to DP, it is not equivalent to the DP mathematical combination because our algorithm generates only

optimal solutions but from different valuation directions, besides the DP give only exact solution

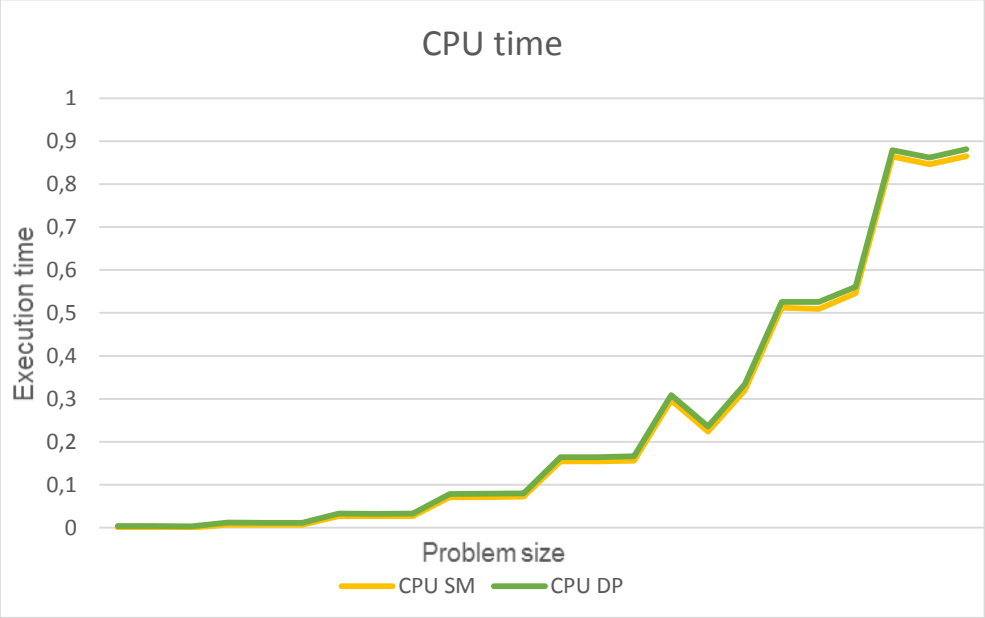


Figure 4.2 Reports the progress of the computational time in terms of the problem size.

Based on comparison of CPU time we notice that, the CPU rises relatively to SM and DP methods. Besides, the execution time for both methods are shown that CPU time are very close to each other.

4.4.2 Large size items

Now we consider large instances, where the number of items n is within {50,500}. For each problem size, 10 instances were performed. Table 4.2 reports results for the SM in terms of a number of performance measures.

Table 4.2: Experimental results for the SM with large size items

n	# $instances$	CAPACITY RATIO	SM	DP	ERROR	CPU SM	CPU DP
		Min = 0.2	128.0	132.1	0,03103709	3,566	0,203
50	10	Avr = 0.6	228.9	232.8	0,01675258	6,235	0,175
		Max = 0.8	253.2	258.2	0,01936483	10,183	0,211
		Min = 0.2	269.9	277.9	0,02878733	25,125	0,358
100	10	Avr = 0.6	440.4	483.4	0,08895325	25,512	0,361
		Max = 0.8	488.5	539.0	0,09369202	25,123	0,326
		Min = 0.2	497.4	536.1	0,07218802	360	0,304
200	10	Avr = 0.6	889.0	947.9	0,06213736	360	0,312
		Max = 0.8	995.4	1060.2	0,06112054	360,23	0,315
		Min = 0.2	789.0	814.7	0,03154535	1300,5	0,307
300	10	Avr = 0.6	1300.2	1422.8	0,08616812	1290,43	0,311
		Max = 0.8	1403.4	1588.8	0,11669184	1300	0,328
		Min = 0.2	997.2	1077.6	0,07461024	3600	0,313
400	10	Avr = 0.6	1565.3	1882.6	0,1685435	3673	0,386
		Max = 0.8	1707.3	2097.7	0,1861086	3734	0,411
		Min = 0.2	1097.7	1351.8	0,18797159	3923,34	0,367
500	10	Avr = 0.6	2090.3	2364.8	0,11607747	3925,21	0,45
		Max = 0.8	2249.3	2630.6	0,14494792	3910,76	0,492

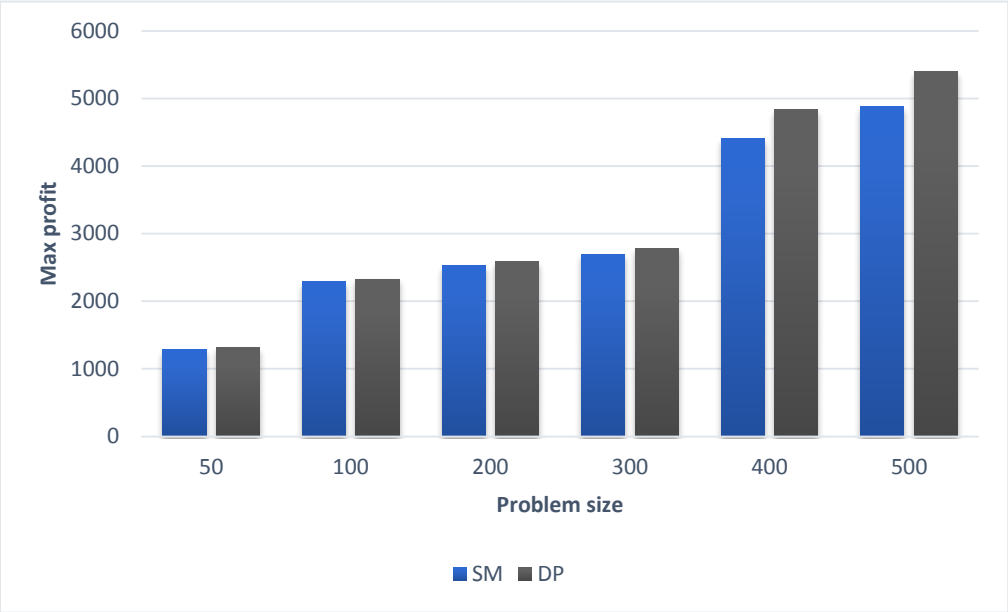
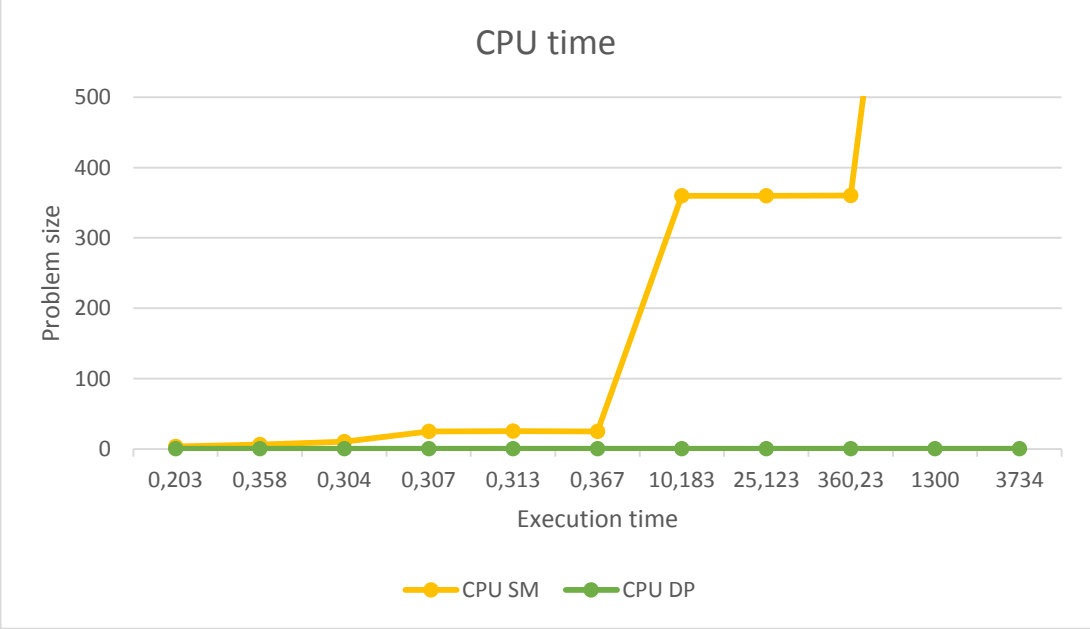


Figure 4.3 Histogram of SM algorithm and the DP of the same instances obtained statically. Results are about 90% of the optimal values.

Results showed that the SM is 90% near to DP, we notice that in problem size 50 to 300 SM equivalent to DP besides where problem size 400 to 500 SM are very close to DP, note that is in small instances.



Figuer 4.4 The CPU time behavior in terms of small problem sizes with large size items

We notice that the SM have an increase behavior in CPU time when the size of problem increase beside the DP have equivalent behavior in large sizes.

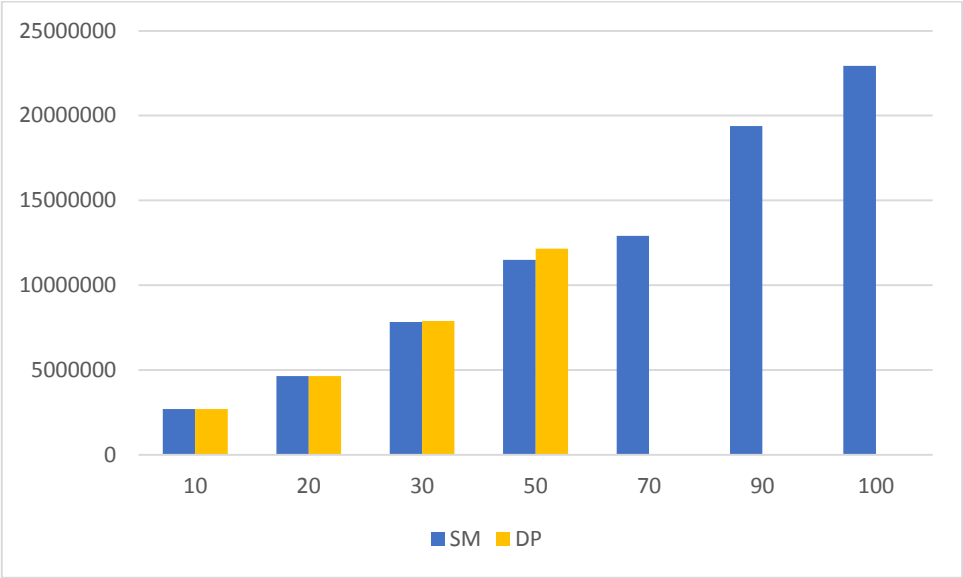
Results show that the SM, the CPU increase proportionally to the problem size. In what follows, we define the rest of used measures and explain this choice.

4.4.3 Large instances

Now we consider large instances, where the number of items n is within {10,100}. For each problem size, 10 instances were performed where the value of item $Vi \approx 1000000$,and weight $Wi \approx 1000000$. Table 4.3 reports results for the SM in terms of a number of performance measures, which are respectively: the number of loaded items, optimal solution of SM, and DP, and the computational time. All instances has been performed for both utility functions presented in the earlier chapter.

Table 4.3: Experimental results for the SM with large size items

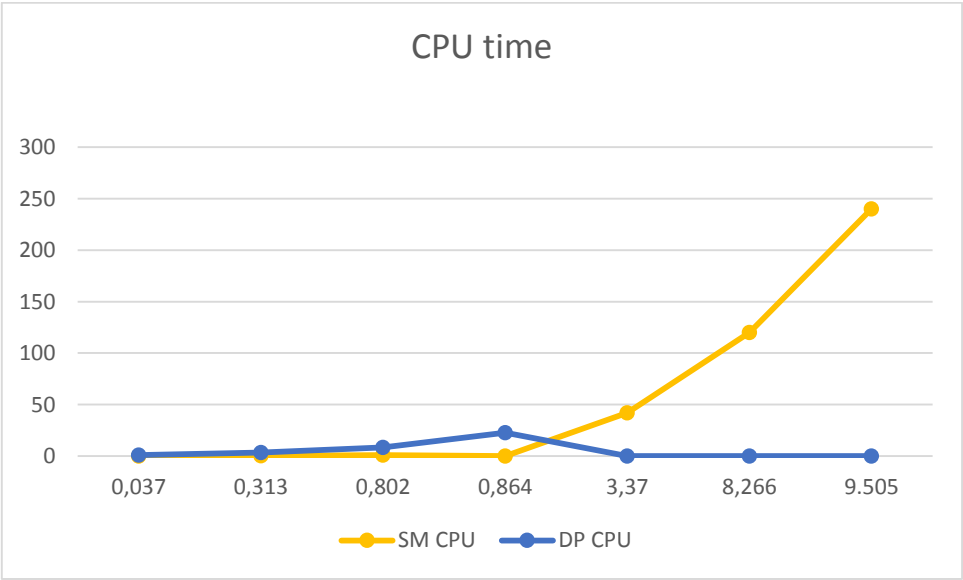
N	SM	DP	SM CPU	DP CPU
10	2698904	2698904	0,037	0,802
20	4637607	4637607	0,313	3,37
30	7890512	7890512	0,864	8,266
50	12151918	12151918	9.505	22,592
70	12914420	/	42	/
90	19376829	/	120	/
100	22934736	/	240	/



Figuer 4.5 Progress of the number of possible collections of items in terms of problem sizes with large instances.

We generates SM various possibility of large instances items where $Vi \approx 1000000$,and weight $Wi \approx 1000000$. Each possibility of size represent a collection of different items.

Evidently, the SM is better than DP in large instances items as shown in figure 4.5. However, the SM results showed that is good and better in large instances when we compare to DP where is stopped in size 50.



Figuer 4.6 the CPU time behavior in terms of large instances problem sizes

Results showed that the SM have polynomial time when it executed; SM algorithm is fast than DP in large instances, besides DP stopped the execution in time where the instances size increase, the DP is exponential time complexity, algorithms which have polynomial time complexity grow much faster than exponential algorithms in large instances.

To conclude, we can say that sieve algorithm has proved to be efficient in solving the knapsack problem in large instances. Compared with optimal results in small size items, we reached almost the same overall profit we notice that our results are about 99% of the optimal values. In the other side of large size items results are 80% of the optimal values, besides in large instances SM proved that is more efficient and polynomial then DP.

4.5 Conclusion

We presented in this chapter the empirical study. Our results showed the optimality in filling the knapsack for different problem sizes. This demonstrate the effectiveness of sieve approach in solving 0/1 KP. However, we notice that in large instances sieve method is polynomial time complexity where grow much faster.

General Conclusion and perspectives

The Knapsack problem is a combinatorial optimization problem where one has to maximize the benefit of objects in a knapsack without exceeding its capacity. It is an NP-complete problem and as such an exact solution for a large input is practically impossible to obtain.

The proposed approach for solving optimization problems sieve method is inspired from the sieving operation to give a new alternative for existing approaches as genetic algorithm, and ant colonies. This approach consists random and iterative batches of solutions in a great set. It eliminates bad items and makes good items in a small set by moving them according to appropriate speed. Was provide a mathematical formulation for this approach as a general algorithm. Since the goal of the proposed method is to find the best solutions in reasonable time especially for great problems sizes,

The objective of this thesis is a survey about sieve method and its application we chose a 0/1 knapsack problem and we give a comparable results with the existent state of art. Using the VB language.

Though the good results of this approach in small size problem and in large instances with small sizes, but it stills missing a convenient behavior with the large size in case of 0/1 knapsack problem because we saw clearly the decreasing of its performance on it.

However, sieve method shown a great performants in large instances items in optimal results and CPU time better then DP.

As a consequence, we aim in the near future to optimize sieve method in order to improve the results that already obtained such as the genetic algorithms or another methods, besides applicate SM on another knapsack problems such Multidimensional KP or Multiple-KP, or dynamic knapsack problem.

In the other hand, we will focus on other technics to accelerate the process in the search more than now, specifically how to pick the best local optimum solution quickly.

Bibliography

Books

[1] A.Carmen Cojocaru, M. Ram Murty, An Introduction to Sieve Methods and Their Applications,
[2] A.Lew, H.Mauch, Dynamic Programming: A Computational Tool, Springer Science & Business Media, 9 Oct. 2006, p 5

[3] Ding-Zhu Du, Panos M. Pardalos, Handbook of Combinatorial Optimization, 1999 p 302.

[4] F. Hillier and G. L*i*eberman, Introduction to Operation research, seventh edition, New York 2002.

[5] F.Rothlauf , Design of Modern Heuristics: Principles and Application, Springer Science & Business Media edition, 17 juil. 2011, page 8

[6] M. Dorigo and G. Di Caro. The ant colony optimization meta-heuristic. In Glover F. Corne D Dorigo M., editor, New Ideas in Optimization, pages 11 32. Mc Graw Hill, London, UK, 1999.

[7] M. Dorigo, G. Di Caro, and L.M. Gambardella. Ant algorithms for dicrete optimization. Artificial Life, 5(2):137 172, 1999.

[8] Martello, S. and P. Toth, Knapsack Problems: Algorithms and Computer Implementation, John Wiley & Sons, New York,1990.p 13

[9] M. R. Garey, D. S. Johnson. Computers and Intractability: a guide to the theory of NP-completeness, W.H.Freeman & Co, 1979.

[10] W.Fred. Glover, Gary A. Kochenberger ,Handbook of Metaheuristics, , 11 avr. 2006,p 306

Papers

[11] A.Hemmak, B.Bouderah Sieve Algorithm - A New Method for Optimization Problems, Int. J. Advance. Soft Comput. Appl., Vol. 5, No. 2, July 2013

[12] An Introduction to Genetic Algorithms, Melanie Mitchell, MIT Press, forthcoming 31,39,1995.

[13] Class NP, NP-complete, and NP-hard problems W. Hamalainen November 6, 2006

[14] F.Glover, Future Paths for Integer Programming and Links to Artificial Intelligence,Computers and Operations Research, Pergamon Journals ,Vol. 13, 1986,pp. 533-549 Lid

[15] M. Dorigo and T. Stutzle. The ant colony optimization metaheuristic: Algorithms, applications and advances. In F. Glover and G.A. Kochenberger, pages 251 286. Kluwer Academic Publishers, Boston, 2003.

[16] T.DAVIDOVIC, D.TEODOROVIc, M.SELMIC, Invited survey BEE COLONY OPTIMIZATION PART I: THE ALGORITHM OVERVIEW, . Yugoslav Journal of Operations Research 25 (2015), Number 1, 33–56, May 2014

Thesis

[17] Metaheuristics and combinatorial optimization problems, Gerald Skidmore Scriptor, (Masters) B.S university of Bufflo, 2006

[18] S.BOUAMAMA, Design of a Learning Method for Automatic Data Extraction, Doctorat, setif, 2013.

[19] H.Ben Romdhan,The optimal stopping knapsack problem,Master,Tunis,2010

Website

[20] http://www.cscscientific.com/particle-size/sieves (consulted in 10/04/2017)

[21] https://www.merriam-webster.com/dictionary/sieve (consulted in 10/04/2017)

[22] http://www.swarmintelligence.org/ (consulted 26/03/2017)

[23] http://www.technion.ac.il/en/ (consulted in 14/5/2017)

[24] https://www.en.wikipedia.org/wiki/Branch_and_bound (consulted in 19/02/2017)

[25] https://en.wikipedia.org/wiki/Knapsack_problem (consulted in 10/02/2017)

[26] http://www.cut-the-knot.org/Curriculum/Arithmetic/Eratosthenes.shtml (consulted in 28/04/2017).

[27] https://studylib.net/doc/7950961/an-overview-of-sieve-methods-1-sieves (consulted in 28/04/2017)

[28] http://dspace.knust.edu.gh/bitstream/123456789/4522/1/Kizito%20Essandoh.pdf

Annex

Sieve of Eratosthenes

Introduction

Eratosthenes (276-194 B.C.E) was the director of the famous library; he was the third librarian of Alexandria and an outstanding scholar all around. He is remembered by his measurement of the circumference of the Earth, estimates of the distances to the sun and the moon, and, in mathematics, for the invention of an algorithm for collecting prime numbers. The algorithm is known as the *Sieve of Eratosthenes*. The sieve of Eratosthenes was first described in the work of Nicomedes (280-210 B.C.E), entitled *Introduction to Arithmetic*. [26]

Sieve methods are used in factoring, recognizing primes, finding natural numbers in arithmetic progression whose common difference are primes, or generally to estimate the cardinalities of various sets defined by the use of multiplicative properties. Recall that use of a sieve or sieving is a process whereby we find numbers via searching up to a prescribed bound and eliminate candidates as we proceed until only the desired solution set remains. In other words, sieve theory is designed to estimate the size of sifted sets of integers. For instance, sieves may be used to attack the following open problems, for which sieve methods have provided some advances.[27]

(a) **(The Twin Prime Conjecture)** There are infinitely many primes p such that $p + 2$ is also prime.
(b) **(The Goldbach Conjecture)** Every even integer $n > 2$ is a sum of two primes.
(c) **(The $p = n2 + 1$ Conjecture)**There are infinitely many primes p of the form $p = n2 + 1$.
(d) **(The $q = 4p + 1$ Conjecture)**There are infinitely many primes p such that $q = 4p + 1$ is also prime.
(e) **(Artin's Conjecture)** For any non-square integer, $a \notin \{-1,0,1\}$, there exist infinitely many primes p such that a is a primitive root modulo p.

Overview of sieve Eratosthenes

A prime number is a natural number that has exactly two distinct natural number divisors: 1 and itself.

To find all the prime numbers less than or equal to a given integer *n* by Eratosthenes' method:

1. Create a list of consecutive integers from 2 through n: $(2, 3, 4, \ldots, n)$.
2. Initially, let p equal 2, the smallest prime number.
3. Enumerate the multiples of p by counting to n from $2p$ in increments of p, and mark them in the list (these will be $2p, 3p, 4p, \ldots$; the p itself should not be marked).
4. Find the first number greater than p in the list that is not marked. If there was no such number, stop. Otherwise, let p now equal this new number (which is the next prime), and repeat from step 3.
5. When the algorithm terminates, the numbers remaining not marked in the list are all the primes below *n*.

The main idea here is that every value given to p will be prime, because if it were composite it would be marked as a multiple of some other, smaller prime. Note that some of the numbers may be marked more than once (e.g., 15 will be marked both for 3 and 5).[27]

As a refinement, it is sufficient to mark the numbers in step 3 starting from p^2, as all the smaller multiples of p will have already been marked at that point. This means that the algorithm is allowed to terminate in step 4 when p^2 is greater than *n*. Another refinement is to initially list odd numbers only, (3, 5, . .., *n*), and count in increments of $2p$ from p^2 in step 3, thus marking only odd multiples of *p*. This actually appears in the original algorithm. This can be generalized with wheel factorization, forming the initial list only from numbers coprime with the first few primes and not just from odds (i.e., numbers coprime with 2), and counting in the correspondingly adjusted increments so that only such multiples of p are generated that are coprime with those small primes, in the first place.

Example

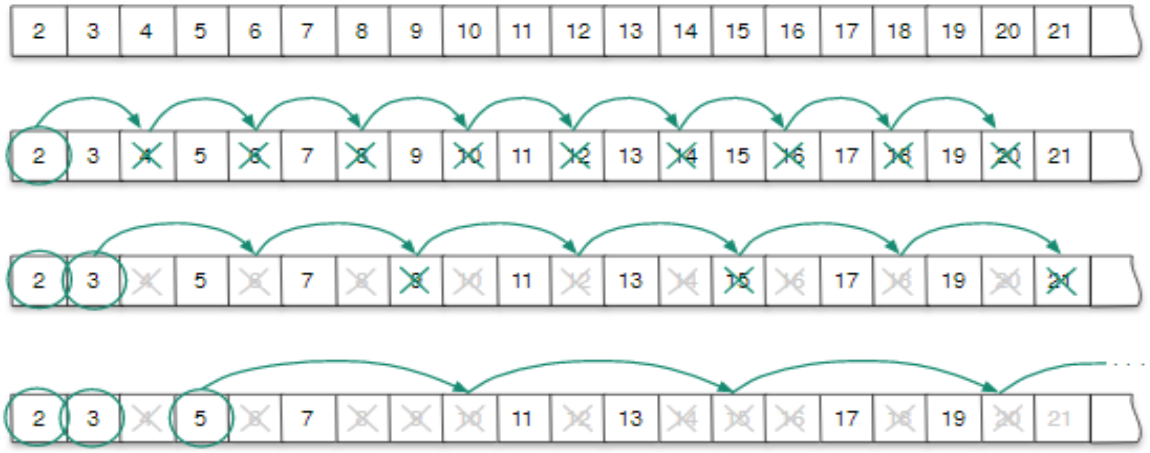


Figure 5.1: To find prime numbers, write the integers starting from 2 on a strip of paper, then systematically cross off multiples of 2, multiples of 3, etc. Until only primes are left.

ملخص

نعالج في هذا العمل مشكلة المعروفة بحقيبة الظهر 0-1 والتي ثبت أنها من المسائل الصعبة، حيث يكون لدينا مجموعة من العناصر، لكل منها وزن وقيمة، وعلينا أن نحدد العناصر التي تُدرج في الحقيبة، بحيث يكون مجموع أوزانها أقل من أو يساوي قدراً معيناً، وأن يكون مجموع القيم أعلى ما يكون.

قمنا بدراسة وتنفيذ نهج الغربلة والذي يقوم على فكرة عملية النخل المستخدمة لفحص الحبوب التي ترجمت للأداة خوارزمية .حيث نقوم بالغربلة بشكل عشوائي وتكراري عدد كبير من المرات للحصول على الحلول الممكنة عن طريق دفعات حيث يولد بشكل عشوائي وتكراري عدد كبير من الحلول الممكنة عن طريق دفعات .قمنا بتطبيق 0/1 خوارزمية الغربال على مشكلة حقيبة الظهر وتمت مقارنة النتائج لإظهار فعالية الخوارزمية .وأظهرت النتائج أن الطريقة المقترحة يمكن أن توفر أفضل الحلول في حالة القيم الكبيرة تم اختبار أداء الطريقة على مجموعة من البيانات المعروفة.

كلمات مفتاحية: مشكلة حقيبة الظهر، المشاكل التوافقية، نهج الغربال، خوارزمية

Abstract

In this study we deal with 0/1 knapsack problem, which is one of combinatorial problems known as NP-hard, where one, has to maximize the benefit of objects in a knapsack without exceeding its capacity. We have done a survey and implementation of this approach, known as Sieve approach is based on the sieving operation idea used to sift grains by translating it on an algorithmic tool. Where generates randomly and iteratively a great number of feasible solutions by batches. The implementation is done on a 0/1 knapsack problem and the results are compared to show the approach efficiency. The findings show that our proposed method can give better solutions in large instances items than other method in combinatorial optimization. The performance of our approach has been tested on well-known dataset.

Keywords: knapsack problem, combinatorial problems, sieve method, optimization.

Résumé

Dans cette étude, nous traitons le problème du 0/1 sac-à-dos prouvé NP-complet, où l'on doit maximiser le bénéfice des objets dans un sac à dos sans dépasser sa capacité. Nous avons fait une étude et l'implémentation de cette approche, connue sous le nom de l'approche Sieve est une méthode métaheuristique, est basée sur l'idée de tamisage utilisé pour tamiser des grains en le traduisant sur un outil algorithmique. Où génère de manière aléatoire et itérative un grand nombre de solutions réalisables par lots. L'implémentation se fait sur le problème de 0/1 sac à dos et les résultats sont comparés pour montrer l'efficacité de l'approche. Les résultats montrent que notre méthode proposée peut donner des meilleures solutions dans de grands cas des valeurs d'objet que d'autres méthodes en optimisation combinatoire. La performance de notre approche a été testée sur un ensemble de données bien connu.

Mots Clés :Problème sac à dos, problème d'optimisation, sieve, optimisation