REPUBLIQUE ALGERIENNE DEMOCRATIQUE ET POPULAIRE
MINISTERE DE L'ENSEIGNEMENT SUPERIEUR ET DE LA RECHERCHE SCIENTIFIQUE
UNIVERSITE MOHAMED BOUDIAF - M'SILA

FACULTE Mathématique et Informatique

DEPARTEMENT Informatique

N° :……..

DOMAINE : Mathématique et Informatique

FILIERE : Informatique

OPTION : OUTILS ET METHODES DE L'INFORMATIQUE DECISIONNELLE

Mémoire présenté pour l'obtention
Du diplôme de Master Académique

Par: LOUANAS Salma

Intitulé

A Hybrid Metaheuristic for the Minimum Weight Dominating Set Problem

Soutenu devant le jury composé de :

PROF.GASMI Abdelkader	Université de M'sila	Président
DR.BOUAMAMA Salim	Université de M'sila	Rapporteur
DR.MOUSSAOUI Adel	Université de M'sila	Examinateur

Année universitaire : 2016 /2017

A C K N OW L E D G E M E N T S

In The Name of **ALLAH**, the Most Beneficent, the Most Merciful.

All praise belongs to **ALLAH** alone, and blessings and peace be upon

the final Prophet

At first a many thanks to my **Parents** and my family and my friends

Many thanks to my supervisors **DR. BOUAMAMA Salim**

for his supports, and advises.

Many and special thanks for **PROF.BOUDERAH Brahim**

for his encouragement, support and advises

Contents

Contents……I

List of figures……..III

List of tables……...IV

Preface……..1

Chapter 1: Combinatorial Optimization

1. Introduction……3
2. Optimization problem……..5
   - 2.1. Definition of an optimization problem……..5
3. Statement of an Optimization Problem……5
   - 3.1. Design vector……5
   - 3.2. Design constraints……..6
   - 3.3. Objective function……..6
4. Classification of Optimization Problems……7
5. Computational Complexity……...10
   - 5.1. Complexity classes……..10
6. Optimization methods……...11
   - 6.1. Exact methods……..11
   - 6.2. Approximate methods……12

Chapter 2: Minimum Weight Dominating Set Problem

1. Introduction……14
2. Fundamentals of Graph Theory……15
   - 2.1. Graph……15
   - 2.2. Definitions and Notations of Graphs……...15
   - 2.3. Some Common Type of Graphs……16
   - 2.4. Graphs representations……19
3. Minimum Weight Dominating Set Problem……...….21
   - 3.1. Problem statement……21
   - 3.2. Example……22
   - 3.3. MWDSP complexity……23
   - 3.4. Approximation algorithms for MWDSP……..23

Chapter 3: Carousel Greedy Algorithm and Local Search for The Minimum Weight Dominating Set Problem

1. Greedy Algorithms……24
2. Carousel Greedy Algorithm……...24
3. Local Search……...25
4. Greedy heuristic for the MWDS problem……..25
   4.1. Pseudo code of Greedy algorithm for the MWDS problem……26
5. Modified Carousel Greedy algorithm for the MWDS problem……27
6. Elimination of redundant vertices……..28
7. Local Search for the MWDSP……29
   7.1. Configuration Checking……...29
   7.2. Two-Level Configuration Checking……31
   7.3. The Selection Vertex Strategy……33
   7.4. CC^2FS Algorithm……34

Chapter 4: Analyze of results

1. Introduction……36
2. Experimental evaluation……..36
   2.1. Problems instances……36
   2.2. Numerical results……...37

Conclusion and Future Work……...40

Bibliography……...41

List of Figures

1.1 The process of the optimization cycle…….4
1.2 Feasible region in a two-dimensional design space. Only inequality constraints are Present……..6
1.3 Design space, objective functions surfaces, and optimum point…….7
1.4 Classification of optimization problems……..8
2.1 A simple graph……...15
2.2 Null Graph…….16
2.3 Trivial Graph…….17
2.4 Undirected graph……...17
2.5 Directed Graph……..17
2.6 Connected Graph……...18
2.7 Disconnected Graph……..18
2.8 Regular Graph……19
2.9 Complete Graph…….19
2.10 Two different graphs that is isomorphic……..20
2.11 Adjacency matrix representation…….20
2.12 Adjacency list representation……...21
2.13 An illustrative example of MWDSP……22
4.1**:** comparison with Raka-ACO on SMPI instances……38
4.2**:** comparison with Raka-ACO on LPI instances……39

List of Tables

Table 4.1: Comparison of results for SMPI for Type1 problems……..37
Table 4.2: Comparison of results for large problems for Type1 problem……...38

Preface

Optimization is a scientific discipline that is concerned with obtaining the best result under given circumstances. Many challenging applications in business, economics, and engineering can be modeled as optimization problems. In fact, real-life problems solving are often complex and difficult to solve to which optimization plays a key role in finding feasible solutions to these problems.

Combinatorial optimization is a lively field of applied mathematics, combining techniques from combinatorics, linear programming, and the theory of algorithms, to solve optimization problems over discrete structures. A combinatorial optimization problem is defined over a set of instances (admissible input data); each instance has a finite set of feasible solutions associated with it. Given an instance, the aim is to determine solutions that maximize or minimize a certain measure function.

The minimum dominating set (MDS) problem aims at finding a dominating set of minimum cardinality in a given undirected graph $G(V; E)$. Hereby, a subset of vertices S $\in$V is called a dominating set if each vertex in V is contained in S or adjacent to some vertex in S. The MDS problem is NP-hard and has various practical applications such as, for example, in wireless ad-hoc networks.

In this thesis, we study a variant of the MDS problem in undirected graphs with a positive integer weight assigned to each vertex, namely the minimum weight dominating set (MWDS) problem. This problem was also shown to be an NP-hard. Apart from applications in wireless ad-hoc networks, the MWDS problem has also been used for modeling query-focused multi-document summarization.

The aim of this thesis is to apply the new algorithm Modified Carousel Greedy introduced by Carmine Cerrone, Raffaele Cerulli, Bruce Golden[8] and new local search algorithm for the MWDS problem, which is based on two new ideas introduced by YiyuanWang, Shaowei Cai, Minghao Yin[16]in the hope to enhance the quality of produced solutions. The concept of greedy heuristics is used to generate initial candidate solutions with good results. Moreover, performances of our approach have been tested on well-known dataset. This dissertation is structured as follows.

Chapter 1 reviews some the basic concept of combinatorial optimization problems algorithms, 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.

Chapter 2 gives a general introduction to the basic terminology of graph theory pertinent to our study and a deep description of the minimum weight dominating set problem.

Chapter 3 explains the principal basics of a Modified Carousel Greedy Algorithm and Local Search that have been applied to tackle the MWDS problem.

Chapter 4 reports the results of an experimental performance evaluation of these approaches on several well-known benchmark datasets and the last chapter concludes the thesis.

CHAPTER 1
COMBINATORIAL OPTIMAZATION

Contents

1. Introduction……...3
2. Optimization problem……5
   - 2.1. Definition of an optimization problem……..5
3. Statement of an Optimization Problem…….....5
   - 3.1. Design vector……5
   - 3.2. Design constraints…….....6
   - 3.3. Objective function…….....6
4. Classification of Optimization Problems……7
5. Computational Complexity……10
   - 5.1. Complexity classes……..10
6. Optimization methods……11
   - 6.1. Exact methods…….....11
   - 6.2. Approximate methods……12

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[23].

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.

For the improvement of the heuristic approaches, techniques of local search have been used. They perform a search of the solutions that are "close" to a given input solution. The set of solutions explored is defined the neighborhood of the input. Local search techniques are able to reach a local optimum of the area explored, but, since they do not have a complete vision of the search space, fail in moving towards other, maybe better, local optima.

To avoid this limit of local search techniques, in the recent years metaheuristic algorithms have been created. They consist in performing intelligent search of the space of solutions, starting with one or more input solutions, and improving them through intense search in the promising areas and diversification for moving towards appealing areas. Many different techniques have been applied, leading to interesting results for many Combinatorial Optimization problems [19].

Optimization involves the study of optimality criteria for problems, the de-termination of algorithmic methods of solution, the study of the structure of such methods, and computer experimentation (implementation) with methods both under trial conditions and on real life problems. The process of optimization may be represented schematically as in Figure 1.1. We begin by understanding and analyzing the real problem in order to determine what to optimize and identify the relevant and essential objects that are present in the problem for which to create a model. Then, we choose and design an algorithm or solution technique to apply to it.

After the developed algorithm is implemented, the two phases of verification and validation are necessary for reviewing the model design according to the objectives and specifications previously defined in the analyze process. The verification ensures that the computer implementation is actually carrying out the algorithm as it is sup-posed to. Validation and sensitivity analysis is the process of making sure that the model or solution technique is appropriate for the real situation and looks at the effect of the specific data on the results. It is here that the loop is completed and we finally compare the results we are obtaining with the real situation. Are the results appropriate? Do they make sense? Does the model need to be modified, or another solution technique chosen? If so, then the loop begins again [20].

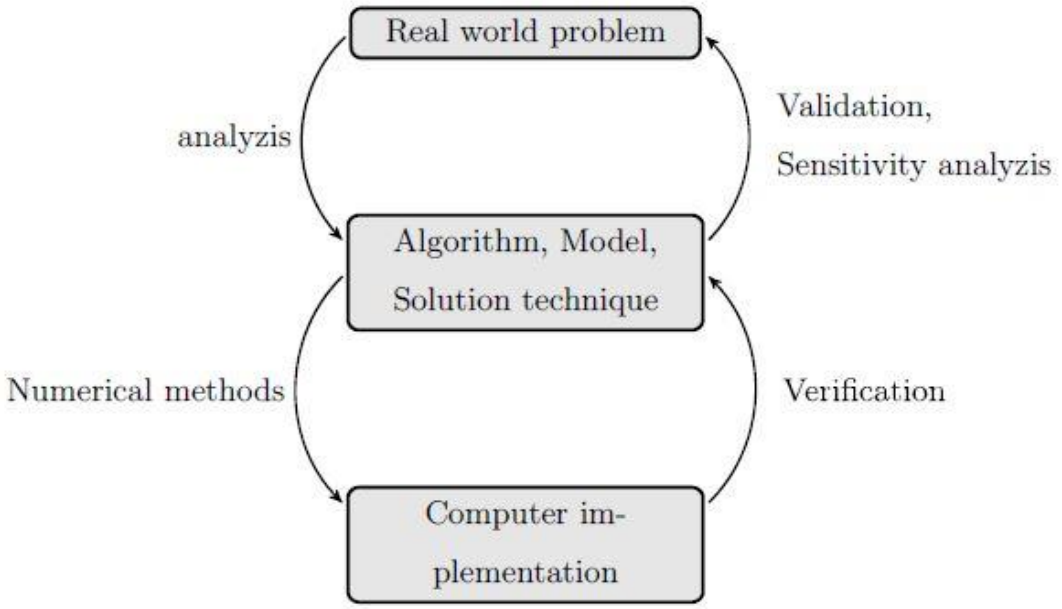


Figure 1.1: The process of the optimization cycle.

2. Optimization Problem

Optimization problems are common in many disciplines and various domains. In optimization problems, we have to find solutions which are optimal or near-optimal with respect to some goals. Usually, we are not able to solve problems in one step, but we follow some process which guides us through problem solving. Often, the solution process is separated into different steps which are executed one after the other. Commonly used steps are recognizing and defining problems, constructing and solving models, and evaluating and implementing solutions [15].

2.1. Definition of an optimization problem

In applied mathematics and theoretical computer science, most optimization problems can be mathematically expressed as the following generic form. [17]

$$\textbf{minimize } f_m(x), \quad (m = 1, \dots, M), \qquad (1)$$
$$x \in s$$
$$\textbf{Subject to } \Phi_j(x) = 0, \quad (j = 1, \dots, J), \qquad (2)$$
$$\psi_k(x) \leq 0, \quad (k = 1, \dots, K), \qquad (3)$$

Where $f_m(x)$, $\Phi_j(x)$ and $\psi_k(x)$ are functions of the design vector x=$(x_1, \dots, x_i, \dots, x_n)^T$ of which the components xi are called design or decision variables, and they can be real continuous, discrete or a mixture of these two. The functions$f_m(x)$, $m = 1, \dots, M$ are called the objective functions, and in the case of M = 1 there is only a single objective. The objective function is sometimes called the cost function or energy function in literature. It is worth noting that it is always possible to transform a minimization problem to maximization problem by proper sign manipulation. The space spanned by the decision variables is called the search space 8, while the space formed by the objective function values is called the solution space. The equalities for $\Phi_j(x)$ and inequalities for $\psi_k(x)$ are called constraints [18].

3. Statement of an Optimization Problem

3.1. Design vector

Any system is described by a set of quantities, some of which are viewed as variables during the design process, and some of which are preassigned parameters or are imposed by the environment. All the quantities that can be treated as variables are called design or decision variables, and are collected in the design vector *x* [1].

3.2. Design constraints

In practice, the design variables cannot be selected arbitrarily, but have to satisfy certain requirements. These restrictions are called design constraints. Design constraints may represent limitation on the performance or behavior of the system or physical limitations. Consider, for example, an optimization problem with only inequality constraints, i.e. $\psi_k(x) \leq 0$. The set of values of x that satisfy the equations $\Phi_j(x) = 0$ forms a hypersurface in the design space, which is called constraint surface. In general, if n is the number of design variables, the constraint surface is an $n - 1$ dimensional surface. The constraint surface divides the design space into two regions: one in which $\psi_k(x) < 0$ and one in which $\psi_k(x) > 0$. The points x on the constraint surface satisfy the constraint critically, whereas the points x such that $\psi_k(x) > 0$, for some j, are infeasible, i.e. are unacceptable, see Figure 1.2.

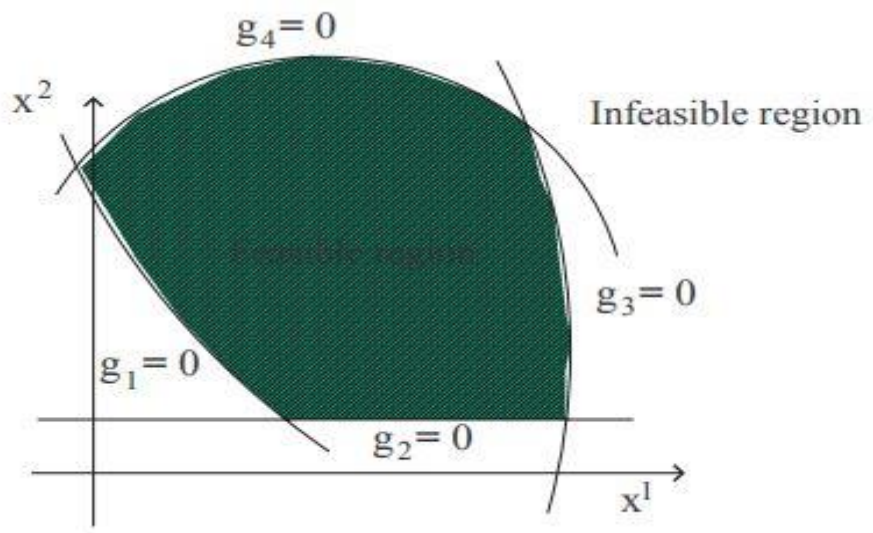


Figure 1.2: Feasible region in a two-dimensional design space. Only inequality constraints are present [1] .

3.3. Objective function

The classical design procedure aims at finding an acceptable design, i.e. a design which satisfies the constraints. In general there are several acceptable designs, and the purpose of the optimization is to single out the best possible design. Thus, a criterion has to be selected for comparing different designs. This criterion, when expressed as a function of the design variables, is known as objective function. The objective function is in general specified by physical or economical considerations. However, the selection of an objective function is not trivial, because what is the optimal design with respect to a certain criterion may be

unacceptable with respect to another criterion. Typically there is a tradeoff performance–cost, or performance–reliability, hence the selection of the objective function is one of the most important decisions in the whole design process. If more than one criterion has to be satisfied we have a multiobjective optimization problem that may be approximately solved considering a cost function which is a weighted sum of several objective functions.

Given an objective function f(x), the locus of all points x such that f(x) = c forms a hypersurface. For each value of c there is a different hypersurface. The set of all these surfaces are called objective function surfaces.

Once the objective function surfaces are drawn, together with the constraint surfaces, the optimization problem can be easily solved, at least in the case of a two dimensional decision space, as shown in Figure 1.3. If the number of decision variables exceeds two or three, this graphical approach is not viable and the problem has to be solved as a mathematical problem. Note however that more general problems have similar geometrical properties of two or three dimensional problems. [1]

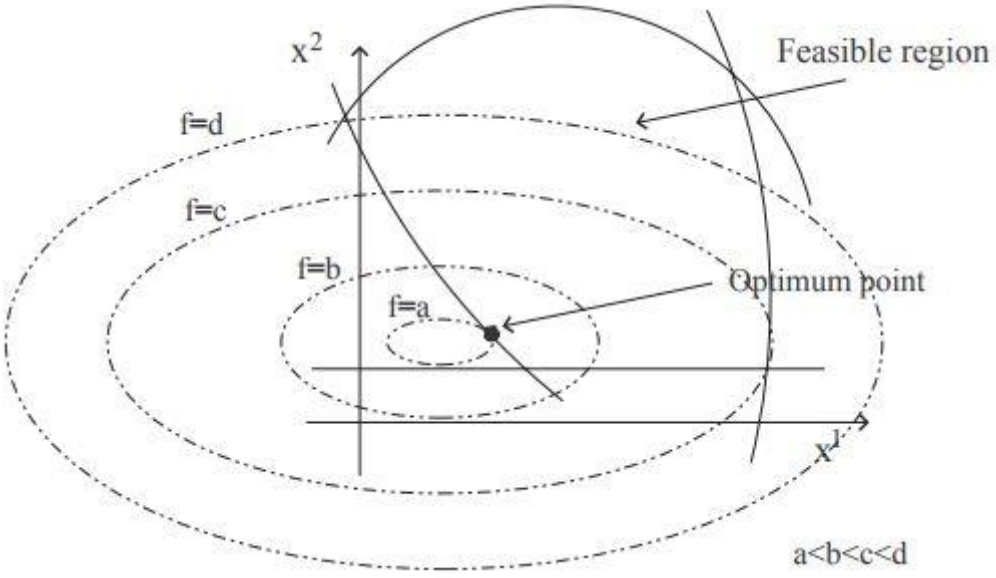


Figure 1.3: Design space, objective functions surfaces, and optimum point. [1]

4. Classification of Optimization Problems

The general optimization problems can be classified as shown in Figure 1.4. We have discussed the objective classification (single or multiple) and the objective type (maximization or minimization) in an earlier section. In case of multiple objectives, the objectives usually contradict each other. If they do not, the multiple objectives can be converted into a single-objective problem. The problem classification (in the next page)

indicates whether the problem contains constraints or not. Some people believe that there are no unconstrained optimization problems in the real world, as these all will have either constraint functions or variable bounds (upper or lower) or both. The study of unconstrained problems is very important since many optimization algorithms solve constrained problems by converting them into an unconstrained or a sequence of unconstrained problems. In addition, several unconstrained optimization techniques can be extended in a natural way to provide and motivate solution procedures for constrained problems.

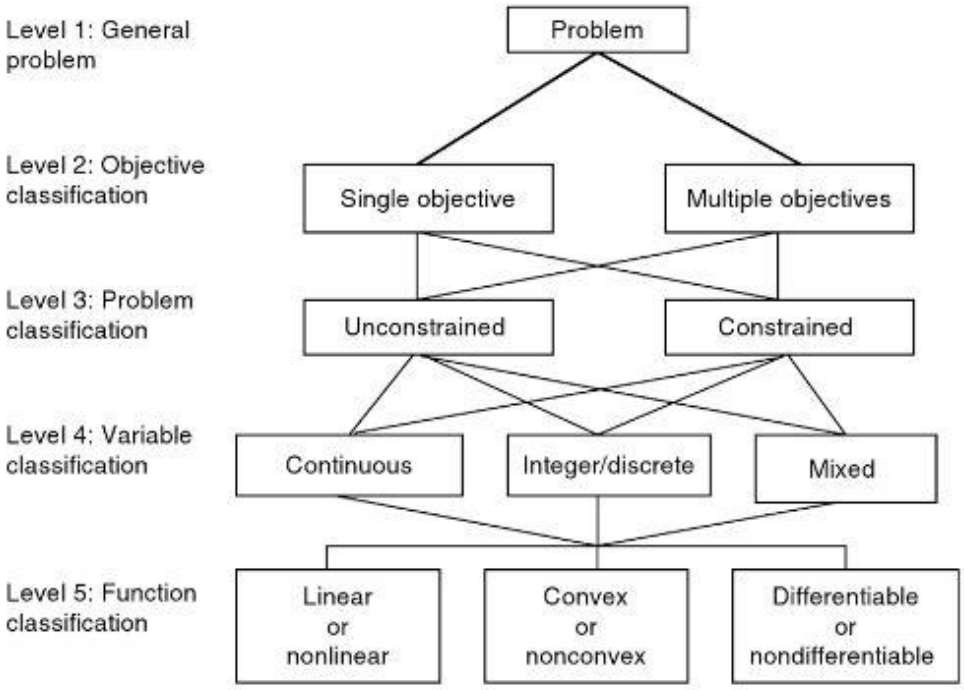


Figure 1.4: Classification of optimization problems

We have already discussed the variable classification as real, integer, or mixed integer. However, many practitioners recognize them as continuous, integer, discrete, or mixed. In problems with continuous (real) variables, we are generally looking for a set of real numbers. The optimization problem with integer or discrete variables is termed a combinatorial problem. In combinatorial problems, we are looking for an object from a finite or infinite set—typically an integer, set, permutation, or graph. These two kinds of problems generally have quite different flavors, the methods for solving them have become quite divergent.

The function classification mainly deals with functions' mathematical properties, which are very important from the solution approach point of view. The objective or constraint functions may be either linear, nonlinear, or both. If all the functions are linear in a

given model, we call it a linear programming model or linear model. If one or more of the functions of a model involve nonlinearity, we call it a nonlinear model. The solution approaches of nonlinear models are quite different and more complex than those of linear models. An unconstrained problem with a single linear objective function does not raise any interest from an optimization point of view. However, unconstrained nonlinear optimization problems attract many interesting research studies and applications.

Convexity is considered as an important property in classical optimization as many optimization techniques/algorithms are developed based on the assumption that the function is convex. In optimization, the solution approaches can be divided into two major groups: (i) those with derivatives and (ii) those without derivatives (derivative free). Differentiability of the function is necessary when using derivative-based techniques. Differentiability is closely related to the continuity of functions. For continuity, differentiability and convexity properties of functions, consult any first year mathematics book. A brief description is also available in works by Hillier and Lieberman (2001) and Bazaraa et al. (1.990).

As an example, the function properties of a single-objective constrained problem with continuous variables could be nonlinear, convex, and differential (see Figure 1.4). In addition to the above general classification, the optimization problem domain also considers function properties such as unimodal versus multimodal, static versus dynamic, and constraint properties such as soft versus hard constraints. A function with only one peak (optimum solution) is known as a unimodal function whereas a function with more than one peak (either local or global optima) is recognized as a multimodal function. If a function changes over time, it is known as a dynamic function. In this book, we restrict ourselves to static functions with a minor variation. The constraints that must be satisfied, in the final solution, are known as hard constraints. Soft constraints are those constraints that can be violated with a certain penalty or under certain conditions. [4]

5. Computational Complexity

Computational complexity refers to the amount of resources required to solve a type of problem by systematic application of an algorithm. Resources that can be considered include the amount of communications, gates in a circuit, or the number of processors. Because the size of the particular input to a problem will affect the amount of resources necessary, measures of complexity will have to take into account this difference. As such, they can be reported as a function of the input size.

An algorithm may be considered to be less complex than others for small inputs, but the same comparison may not hold for larger input sizes. Depending on starting points,

iterative application of algorithms may not cover the same trajectory to the same solution even if the input to the problem is identical. One way of addressing this variability is to compare the upper and lower bounds of complexity (worst and best case scenarios, respectively) and average values. [21] The amount of computational resources is determined by its time and space complexity

- Time complexity describes how many iterations or a number of search steps are necessary to solve a problem or produce its result. Problems are more difficult if more time is necessary.
- Space complexity describes the amount of space (usually memory on a computer) that is necessary to solve a problem. [18]

5.1. Complexity classes

A complexity class is a class of problems grouped together according to their time and/or space complexity

5.1.1. *Class P*

The complexity class P (P stands for polynomial) is defined as the set of decision problems that can be solved by an algorithm with worst-case polynomial time complexity. The time that is necessary to solve a decision problem in P is asymptotically bounded $(for\, n > n_0)$ by a polynomial function$O(n^k)$. For all problems in$P$, an algorithm exists that can solve any instance of the problem in time that is$O(n^k)$, for some *k*. Therefore, all problems in P can be solved effectively in the worst case. As we showed in the previous section that all optimization problems can be formulated as decision problems, the class P can be used to categorize optimization problems [15] .

5.1.2. *Class* NP

This complexity class describes the set of decision problems of which a yes solution can be guessed and verified in polynomial time. Note that NP stands for non-deterministic polynomial time. An equivalent definition of NP is that the set of all decision problems that can be solved by polynomial-time non-deterministic algorithms. Non-deterministic means that no particular rule is followed to make the guess. Besides, P is a subset of NP,$P \in NP$. One of the important open questions in complexity theory is whether class P is equal or not equal classNP. Nevertheless, this has never been proven and it is widely believed that P [18]

5.1.3. Class NP-Complete

Assume that $P \neq NP$, a subset of.$NP \backslash P$ comprising the hardest problems in NP are called $NP - complete$ problems. Formally, A problem is $NP - complete$ if it is in NP and each NP problem can be reduced to it in polynomial time. This class has a significant properly that if any $NP - complete$ problem can be solved in polynomial time, then every problem in NP has a polynomial-time solution, that is, $P = NP$. Despite years of study, no polynomial-time algorithm has ever been found for any $NP - complete$ problem[2].

5.1.4. Class NP-Hard

The class of $NP - Hard$ problems contains problems which are not in NP but any NP, problem can be transformed to them in polynomial time. The name of this class means that such problem is at least as hard as any problem in NP. It is a more general class of problems that comprises, for example, the search and optimization variants of the $NP - complete$ problems. However, the fact that an optimization problem is $NP - hard$ does not imply that all instances are difficult to solve [11]

6. Optimization methods

6.1. Exact methods

An exact method or optimization method for solving an optimization problem is one that is guaranteed to produce, in finite time, a global optimum for this problem and a proof of its optimality, in case one exists, or otherwise show that no feasible solution exists. Globally optimal solutions are often referred to as exact optimal solutions. Among the many exact methods for solving combinatorial optimization problems, we find algorithmic paradigms such as cutting planes, dynamic programming, backtracking, branch-and-bound (together with its variants and extensions, such as branch-and-cut and branch-and-price), and implicit enumeration. Some of these paradigms can be viewed as tree search procedures, in the sense that they start from a feasible solution (which corresponds to the root of the tree) and carry out the search for the optimal solution by generating and scanning the nodes of a subtree of the solution space (whose nodes correspond to problem solutions). [14]

6.2. Approximate methods

Approximate algorithms differ essentially from exact ones as they cannot guarantee to find optimal solutions in finite time or prove that no solutions exist in the case of satisfaction problems. But, for optimization problems, they often find high quality solutions much faster than exact algorithms and are able to successfully attack large instances. Approximate

algorithms can be classified as either constructive or local search algorithms. Approximate algorithms can be classified in two families: specific heuristic and metaheuristic.

6.2.1. Heuristics

Heuristic methods only attempt to yield a good , but not necessarily optimum solution. Nevertheless, the time taken by an exact method to find an optimum solution to a difficult problem, if indeed such a method exists, is in a much greater order of magnitude than the heuristic one (sometimes taking so long that in many cases it is inapplicable). Thus we often resort to heuristic methods to solve real optimization problems. [12]

A heuristic is a technique designed for solving a problem more quickly when classic methods are too slow, or for finding an approximate solution when classic methods fail to find any exact ones ,but they do not guarantee that the best will be found, therefore they may be considered as approximately and not accurate algorithms. These algorithms, usually find a solution close to the best one and they find it fast and easily. Sometimes these algorithms can be accurate, that is they actually find the best solution, but the algorithm is still called heuristic until this best solution is proven to be the best [10] .

6.2.2. Metaheuristics

In computer science, meta-heuristic designates a computational method that optimizes a problem by iteratively trying to improve a candidate solution with regard to a given measure of quality. Metaheuristics make few or no assumptions about the problem being optimized and can search very large spaces of candidate solutions. However, metaheuristics do not guarantee an optimal solution is ever found. Many meta-heuristics implement some form of stochastic optimization. Other terms having a similar meaning as meta-heuristic, are: derivative-free, direct search, black-box, or indeed just heuristic optimizer.Following are properties that characterize most metaheuristics:

- Metaheuristics are strategies that guide the search process. The goal is to efficiently explore the search space in order to find near–optimal solutions.
- Techniques which constitute metaheuristic algorithms range from simple local search procedures to complex learning processes.
- Meta-heuristic algorithms are approximate and usually non-deterministic.
- Metaheuristics are not problem-specific.

Metaheuristics may make few assumptions about the optimization problem being solved, and so they may be usable for a variety of problems. One of the best quote I’ve heard some time

ago to describe the difference between heuristic and meta-heuristic: “A heuristic is a pretty good rule. A meta-heuristic is a pretty good rule for finding pretty good rules”. [10]

CHAPTER 2
MINIMUM WEIGHT DOMINATING SET PROBLEM

Contents

1. Introduction……14
2. Fundamentals of Graph Theory……...15
   - 2.1. Graph……...15
   - 2.2. Definitions and Notations of Graphs……...15
   - 2.3. Some Common Type of Graphs…….16
   - 2.4. Graphs representations……19
3. Minimum Weight Dominating Set Problem……..21
   - 3.1. Problem statement……...21
   - 3.2. Example……...22
   - 3.3. MWDSP complexity……23
   - 3.4. Approximation algorithms for MWDSP……23

1. Introduction

Graph theory may be said to have its beginning in 1736 when EULER considered the (general case of the) Konigsberg bridge problem: Does there exist a walk crossing each of the seven bridges of Konigsberg exactly once? It took 200 years before the first book on graph theory was written. This was “Theorie der endlichen und unendlichen Graphen” (Teubner, Leipzig, 1936) by KÖNIG in 1936. Since then graph theory has developed into an extensive and popular branch of mathematics, which has been applied to many problems in mathematics, computer science, and other scientific and not-so-scientific areas. For the history of early graph theory, Clarendon Press, 1986. There are no standard notations for graph theoretical objects. This is natural, because the names one uses for the objects reflect the applications. Thus, for instance, if we consider a communications network (say, for email) as a graph, then the computers taking part in this network are called nodes rather than vertices or points. On the other hand, other names are used for molecular structures in chemistry, flow charts in programming, human relations in social sciences, and so on.

Graph theory is very much tied to the geometric proprieties of optimization and combinatorial optimization. Moreover, graph theory’s geometric properties are at the core of many research interests in operations research and applied mathematics. Its techniques have been used in solving many classical problems including maximum flow problems, independent set problems, and the traveling salesman problem. Graph theory and Combinatorial Optimization explores the field’s classical foundations and its developing theories, ideas and applications to new problems.

Graph theory has its applications in diverse fields of engineering, Electrical Engineering, Computer Science is used for the study of algorithms, Computer Network the relationships among interconnected computers in the network follows the principles of graph theory, science, Linguistics, Routes between the cities can be represented using graphs. Depicting hierarchical ordered information such as family tree can be used as a special type of graph called tree.

2. Fundamentals of Graph Theory

A graph is a diagram of points and lines connected to the points. It has at least one line joining a set of two vertices with no vertex connecting itself. The concept of graphs in graph theory stands up on some basic terms such as point, line, vertex, edge, degree of vertices, properties of graphs, etc. Here, in this chapter, we will cover these fundamentals of graph theory.

2.1. Graph

A graph G consists of a pair (V, E), where V is the set of vertices and E the set of edges. We write $V\ (G)$ for the vertices of G and $E(G)$ for the edges of G when necessary to avoid ambiguity, as when more than one graph is under discussion. If no two edges have the same endpoints we say, there are no multiple edges, and if no edge has a single vertex as both endpoints, we say there are no loops. A graph with no loops and no multiple edges is a simple graph. A graph with no loops, but possibly with multiple edges is a multigraph. The condensation of a multigraph is the simple graph formed by eliminating multiple edges, which is, removing all but one of the edges with the same endpoints. To form the condensation of a graph, all loops are also removed. We sometimes refer to a graph as a general graph to emphasize that the graph may have loops or multiple edges.[3]

The edges of a simple graph can be represented as a set of two element sets; for example,

$$(\{v_1, \ldots, v_7\}, \{\{v_1, v_2\}, \{v_2, v_3\}, \{v_3, v_4\}, \{v_3, v_5\}, \{v_4, v_5\}, \{v_5, v_6\}, \{v_6, v_7\}\})$$

Is a graph that can be pictured as in figure 1. This graph is also a connected graph: each pair of vertices v, w is connected by a sequence of vertices and edges, $v = v1, e1, v2, e2, \ldots, vk = w$, where v_i and v_{i+1} are the endpoints of edge e_i.

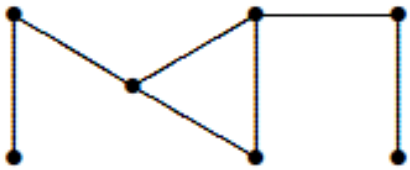


Figure 2.1**:** A simple graph.

2.2. Definitions and Notations of Graphs

- Two vertices v, w are said to be adjacent if there is an edge joining v and w. An edge and a vertex are said to be incident if the vertex is an endpoint of the edge.
- Given a vertex v, the degree of v is defined to be the number of edges containing v as an endpoint.

- A path in a graph G is defined to be a finite sequence of distinct vertices $v_0, v_1, \cdots, v_t$ such that vi is adjacent to $v_i + 1$. (A graph itself can also be called a path.) The length of a path is defined to be the number of edges in the path.
- A cycle in a graph G is defined to be a finite sequence of distinct vertices $v0, v1, \cdots, vt$ such that vi is adjacent to $vi + 1$ where the indices are taken modulo $t + 1$. (A graph itself can also be called a cycle.) The length of a cycle is defined to be the number of vertices (or edges) in the path.
- A graph is said to be connected if for any pair of vertices, there exists a path joining the two vertices. Otherwise, a graph is said to be disconnected.
- The distance between two vertices u, v in a graph is defined to be the length of the shortest path joining u, v. (In the case the graph is disconnected, this may not be well-defined.)
- Let $G = (V, E)$ be a finite graph. A graph G is said to be complete if every pair of vertices in G is joined by an edge. A complete graph on n vertices is denoted by K_n.

2.3. Some Common Type of Graphs

There are various types of graphs depending upon the number of vertices, number of edges, interconnectivity, and their overall structure. We will discuss only a certain few important types of graphs.

2.3.1. Null Graph

A graph having no edges is called a Null Graph

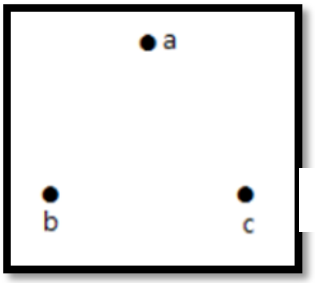


Figure 2.2**:** Null Graph

2.3.2. Trivial Graph

A graph with only one vertex is called a Trivial Graph.

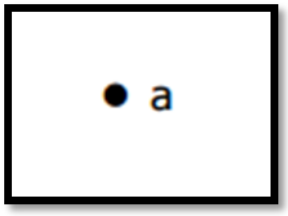


Figure 2.3: Trivial Graph

2.3.3. Undirected Graph

An undirected graph contains edges but the edges are not directed ones

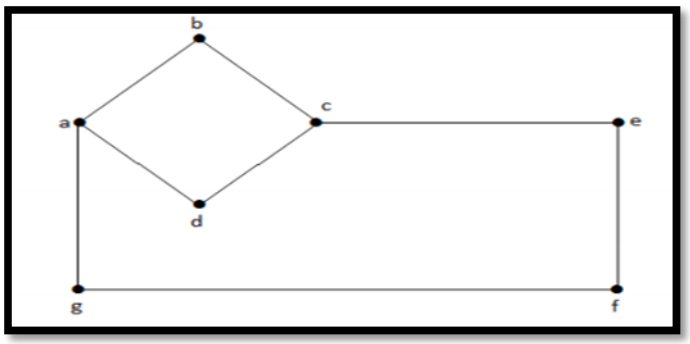


Figure 2.4: Undirected graph

2.3.4. Directed Graph

In a directed graph, each edge has a direction.

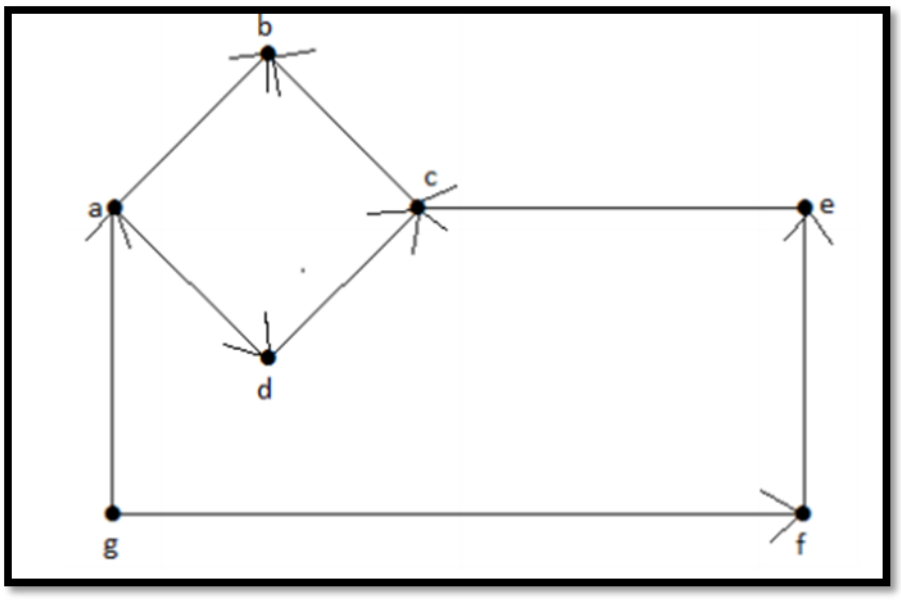


Figure 2.5**:** Directed Graph

2.3.6. Connected Graph

A graph G is said to be connected if there exists a path between every pair of vertices. There should be at least one edge for every vertex in the graph. So that we can say that it is connected to some other vertex at the other side of the edge.

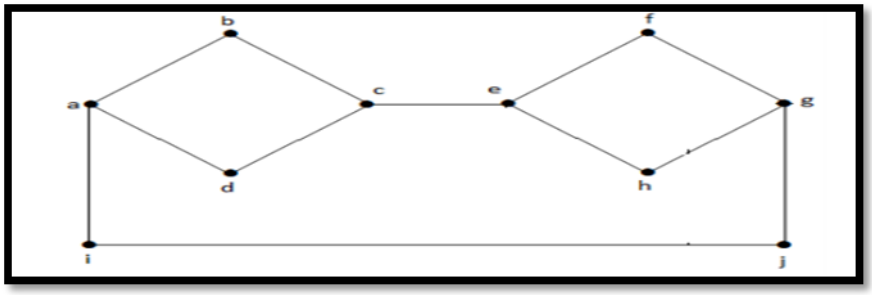


Figure 2.6: Connected Graph

2.3.7. Disconnected Graph

A graph G is disconnected, if it does not contain at least two connected vertices.

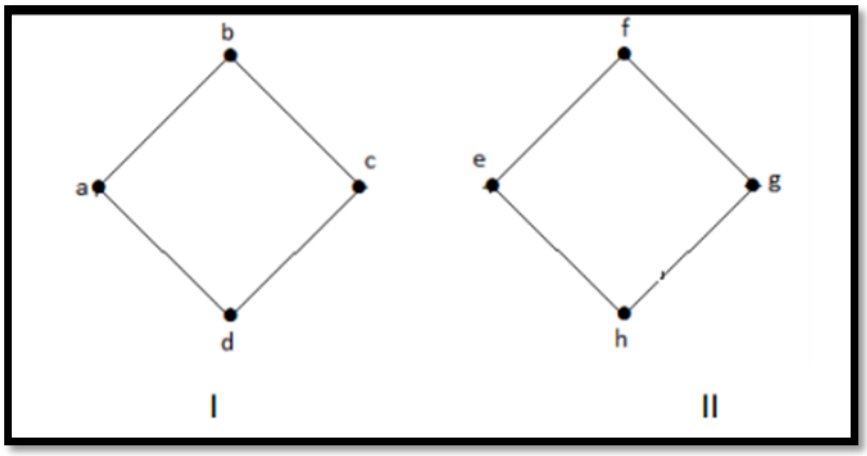


Figure 2.7: Disconnected Graph

2.3.8. Regular Graph

A graph G is said to be regular, if all its vertices have the same degree. In a graph, if the degree of each vertex is ‘k’, then the graph is called a ‘k-regular graph’.

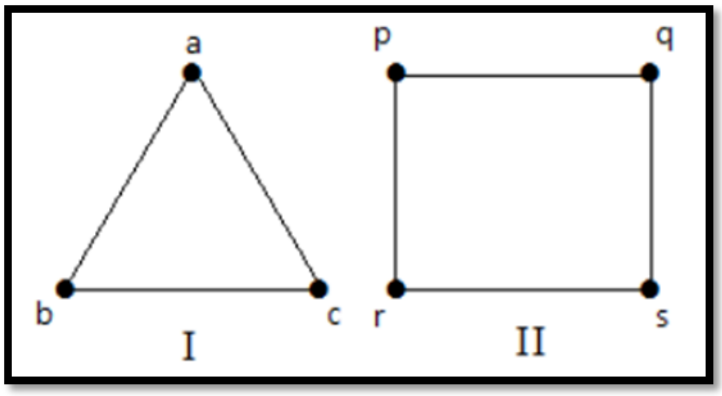


Figure 2.8: Regular Graph

2.3.9. Complete Graph

A simple graph with 'n' mutual vertices is called a complete graph and it is denoted by 'k_n'. In the graph, a vertex should have edges with all other vertices, then it called a complete graph. In other words, if a vertex is connected to all other vertices in a graph, then it is called a complete graph.

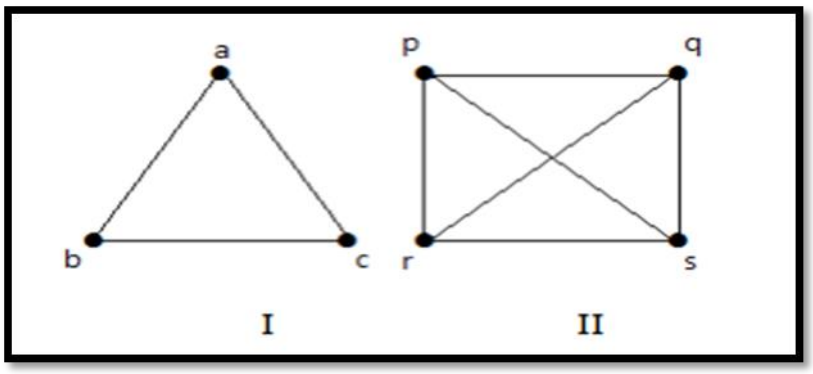


Figure 2.9: Complete Graph

2.4. Graphs representations

There are many ways to represent a graph. We have already seen few ways: we can draw it, as in Figure 2.10 for example, or you can represent it with sets as in $G = (V, E)$ another common representation is with an adjacency matrix or adjacency list.

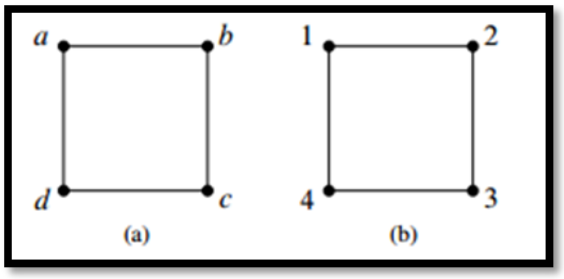


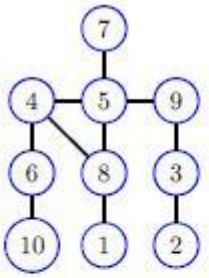
Figure 2.10: Two different graphs that are isomorphic.

2.4.1. Adjacency matrix representation

An adjacency matrix is a square matrix used to represent a finite graph. The elements of the matrix indicate whether pairs of vertices are adjacent or not in the graph.

In the special case of a finite simple graph, the adjacency matrix is a (0, 1)-matrix Figure2.11 with zeros on its diagonal. If the graph is undirected, the adjacency matrix is symmetric Figure2.11. The relationship between a graph and the eigenvalues and eigenvectors of its adjacency matrix is studied in spectral graph theory.

The adjacency matrix should be distinguished from the incidence matrix for a graph, a different matrix representation whose elements indicate whether vertex–edge pairs are incident or not.



v	1	2	3	4	5	6	7	8	9	10
1	0	0	0	0	0	0	0	1	0	0
2	0	0	1	0	0	0	0	0	0	0
3	0	1	0	0	0	0	0	0	1	0
4	0	0	0	0	1	1	0	1	0	0
5	0	0	0	1	0	0	1	1	1	0
6	0	0	0	1	0	0	0	0	0	1
7	0	0	0	0	1	0	0	0	0	0
8	1	0	0	1	1	0	0	0	0	0
9	0	0	1	0	1	0	0	0	0	0
10	0	0	0	0	0	1	0	0	0	0

(a) An undirected graph **(b)** Adjacency matrix representation

Figure 2.11: Adjacency matrix representation

2.4.2. Adjacency list representation

An adjacency list is a list of lists. Each list corresponds to a vertex u and contains a list of edges (u, v) that originate from u.

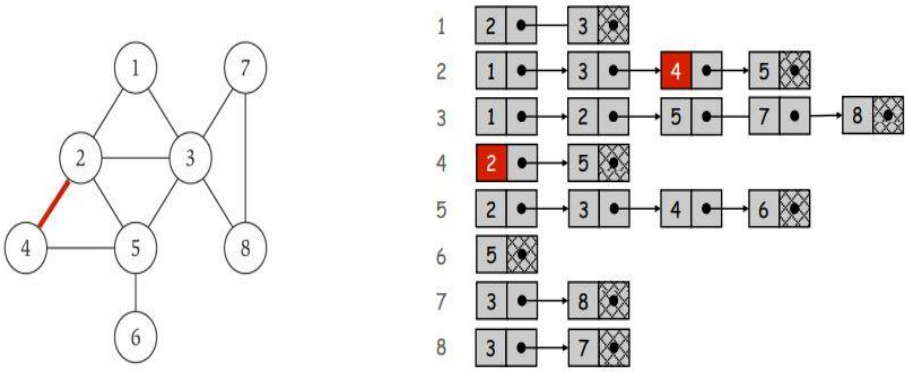


Figure 2.12: Adjacency list representation

3. Minimum Weight Dominating Set Problem

The Minimum Weight Dominating Set (MWDS) problem is an important generalization of the Minimum Dominating Set (MDS) problem with extensive applications.

3.1. Problem statement

A problem instance $(G; \omega)$ of the MWDS problem is a tuple that consists of an undirected graph $G(V; E)$, where V is the set of vertices and E is the set of edges, and a function ω : V $\rightarrow \mathbb{R}^+$ that associates a positive weight value ω (v) to each vertex $v \in V$. Before starting with the technical description of the MWDS problem, we introduce, in the following, notations as well as some definitions. A pair of vertices v and u are said to be adjacent if $(v; u) \in E$. For a vertex $v \in V$, the set of neighbors of v in G is denoted by $N(v) = \{u \in V \mid (v; u) \in E\}$, which is also known as the open neighborhood of v. Similarly, we denote by $N[v] = N(v) \cup \{v\}$ the closed neighborhood of v. The degree $d(v)$ of v is the number of v's neighbors, that is, $d(v) = |N(v)|$. A dominating set $S \subseteq V$ is a subset of the vertices of G such that each vertex $v \in V \setminus S$ has a neighbor in S. Each vertex in S is called a dominator, otherwise it is called a dominated. A dominator dominates (covers) itself and all its neighbors.

The MWDS problem can thus be defined as follows. Any dominating set S of G is a valid solution to the problem. The objective function value of a valid solution S is the sum of the weights of all vertices in S. The optimization objective concerns minimization. The MWDS problem can also be formulated in terms of an integer linear program (ILP).

For convenience, we assume that $V = \{v_1, v_2, \ldots, v_n\}$. A binary variable $x_i \in \{0,1\}$ is assigned to each vertex $v_i \in V$ such that $x_i = 1$ iff v_i belongs to the optimal solution. Moreover, note that a solution S is a dominating set iff $\forall v_i \in V: \sum_{v_j \in N[v_i]} x_i \geq 1$, where $N[v_i]$ is the closed neighborhood of v_i as outlined above. The ILP model for the MWDS problem can then be stated as follows. [6]

$$\textbf{\textit{minimize}} \ \sum_{i=1}^{n} w(v_i)\, x_i$$

$$\textbf{\textit{Subject to}} \ \sum_{v_j \in N[v_i]} x_i \geq 1 \quad \forall v_i \in V$$
$$x_i \in \{0,1\}$$

3.2. Example

In this section we give an illustrative example of MWDSP. As mentioned earlier, a problem instance (G, ω) of MWDSP is composed of an undirected graph $G(V, E)$ and a weight function ω as shown in Figure 4.2. In this case, the number of vertices is equal to the number of edges, that is, $|V| = |E| = 10$. Each node is numbered within the circle with its ID and outside the circle with its weight. For example, the weight of vertex $v = 2$ is $\omega(v) = 84$. The optimal solution is an unordered vertex subset S* = {0, 2, 1, 6}, $S* \in V$, with a total weight of 347.

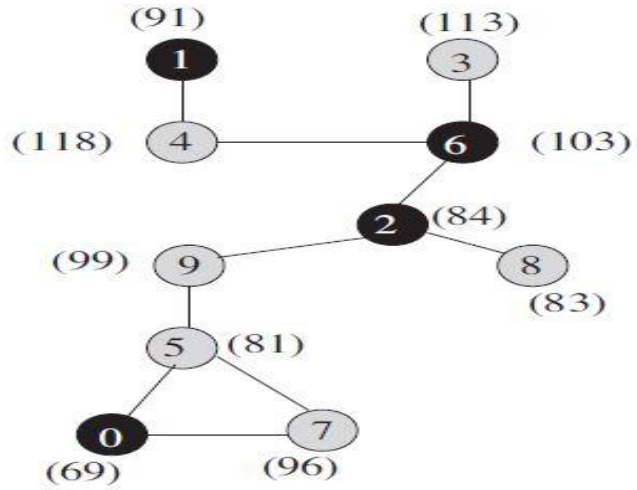


Figure 2.13: An illustrative example of MWDSP

3.3. MWDSP complexity

The MWDSP has been extensively investigated from an algorithmic point of view. It is NP-hard in general. The cardinality version has been shown to be polynomially solvable in several classes of graphs such as cactus graphs and the class of series-parallel graphs. However, to the best of our knowledge, no polynomial time algorithms are known for the MWDSP in these graphs [7]

3.4. Approximate algorithms for MWDSP

MWDS problem has received considerable attention and Therefore, metaheuristics appear as an interesting option to obtain high quality solutions to this problem. These include ant colony optimization approaches [22,13] and a genetic algorithms [13], iterated greedy algorithm [7] .

CHAPTER 3
CAROUSEL GREEDY ALGORITHM AND LOCAL SEARCH FOR THE MWDS PROBLRM

Contents

1. Greedy Algorithms…….…….24
2. Carousel Greedy Algorithm…….…...24
3. Local Search…….…...25
4. Greedy heuristic for the MWDS problem……...25
   - 4.1. Pseudo code of Greedy algorithm for the MWDS problem…...26
5. Modified Carousel Greedy algorithm for the MWDS problem…….……27
6. Elimination of redundant vertices…….……..28
7. Local Search for the MWDSP…….……29
   - 7.1. Configuration Checking……..29
   - 7.2. Two-Level Configuration Checking……31
   - 7.3. The Selection Vertex Strategy…….33
   - 7.4. CC^2FS Algorithm……34
8. CG & CC^2FS Algorithm for the MWDSP…….……35

1. Greedy Algorithms

Greedy algorithms have been developed for a large number of problems in combinatorial optimization.

For many of these greedy algorithms, elegant worst-case analysis results have been obtained. These greedy algorithms are typically very easy to describe and code and they have very fast running times. On the other hand, the accuracy of greedy algorithms is often unacceptable.

When instance size is large, exact solution approaches are generally not practical. However, a wide variety of metaheuristics (e.g., tabu search, genetic algorithms, variable neighborhood search, etc.) have been successfully applied to solve combinatorial optimization problems

2. Carousel Greedy Algorithm

The carousel greedy algorithm is an enhanced greedy algorithm which, in comparison to a greedy algorithm, examines a more expansive space of possible solutions with a small and predictable increase in computational effort. An outline of the carousel greedy algorithm (CG) is as follows:

(i) Create partial solution using a greedy algorithm;

(ii) Use the same greedy algorithm to modify the partial solution in a deterministic way;

(iii) Apply the greedy algorithm to produce complete, feasible solution.

Unlike a typical metaheuristic, CG does not progress from one feasible solution to another, hopefully better, feasible solution. Only one feasible solution, at the end of the algorithm, is identified. In fact, CG does not calculate an objective function until the final step. One can consider the greedy and the CG algorithms to be constructive metaheuristics in contrast to improving metaheuristics [8] .

The carousel greedy can be applied to many types of decision problems. In order to apply CG to a specific decision problem, we assume that there already exists a constructive greedy algorithm for that problem; otherwise, CG may be more difficult to apply. Our primary focus will be on problems involving the minimization or maximization of the cardinality of a set. In particular, we will study the problem of the minimum weight dominating set problem(MWDSP) .the basic steps are provided below:

Step 1: Apply greedy algorithm to obtain a feasible solution.

Step 2: Destruction phase: Remove some elements from the current solution, leaving a partial solution.

Step 3: Construction phase: Apply the greedy algorithm to the partial solution to obtain another
feasible solution.

Step 4: Repeat Steps 2 & 3 until a stopping condition is satisfied.

3. Local Search

Local improvement or local search starts with some feasible solution of the problem and tries to progressively improve it. Each step of the procedure carries out a movement from one solution to another one with a better value. The method terminates when, for a solution, there is no other accessible solution that improves it. [16]

Algorithm 1 The general framework of a local search algorithm

```
1 S := InitFunction() and S*:= S;
2 while not reach terminate condition do
3         if S is better than S* then
4             S*:= S;
5         S := MoveNeighbourPoisition(S);
6 return S*;
```

4. Greedy heuristics for the MWDS problem

Existing greedy heuristics for the MWDS problem build a solution S step-by-step, adding one vertex to S at each construction step. Given a partial solution S that is, a set $S \subset V$ which is not yet a dominating set we can differentiate partition V into three disjoint subsets:

- The vertices in S are called black vertices.
- Vertices that are not contained in S but that are adjacent to at least one vertex from S are called gray vertices.

- The remaining vertices are referred to as white vertices (or uncovered vertices).
  Let $W(S)$ be the set of white vertices with respect to a given partial solution S.

Potluri and Singh [13] proposed four greedy methods for the MWDSP. These greedy methods are based on different greedy functions. A greedy function is used to decide which vertex will be selected in the next iteration. They presented the following four greedy functions

a) $\frac{d_S(v)}{w(v)}$

b) $\frac{W(v)}{w(v)}$

c) $W(v) - w(v)$

d) $\frac{W(v) * d_S(v)}{w(v)}$

Where

$\mathbf{d_S(v)}$ is the number of uncovered neighbours (white neighbours) of vertex v.

W(v) is the total weight sum of white neighbors of a vertex v.

w(v) is the weight of vertex v.

Some experiments of the author's indicate that the first, second and fourth greedy heuristics give similar results. In some cases, one greedy is better than another and vice versa. However, the third greedy heuristic performs
poorly compared to the others.

4.1. pseudo code of Greedy algorithm for the MWDS problem

This procedure takes an empty solution S. At each construction step, the current partial solution S is extended by adding one solution component from $N(S)$ until a complete solution that is, a dominating set is constructed. Remember that N(S) = { $v \in V\backslash S \mid d_S(v) \neq 0$} where the current degree $d_S(v)$ of a vertex $v \in V\backslash S$ with respect to S is defined as

$d_S(v)$= | {$(v,u) \in E \mid u \in$ W(S) [6].

A value for β_0 is randomly generated from a uniform distribution over [0; 1] for each application of greedy procedure. The choice of the next solution component to be added to S at each step is done in the following way. First, each solution component $v \in N(S)$ is rated according to a greedy function $cost(\cdot)$. if $\beta \geq \beta_0$, and $cost(v) = w(v)/ d_S(v)$ else $cost(\cdot)$ is defined as $w(u)/W(v)$.

Algorithm 2 Procedure $GreedyMWDS$(S, β)

1: Input :empty solution S and parameter β
2: $\beta_0 \leftarrow$ random number uniformly distributed over [0,1].
3: N(S) = { $v \in V \backslash S \mid d_S(v) \neq 0$}
4: while $N(S) \neq \emptyset$ do
5: if $\beta \geq \beta_0$ then
6: $v_{best} \leftarrow$ argmin { $w(v)/ d_S(v) \mid \in N(S)$ }.
7: else
8: $v_{best} \leftarrow$ argmin { $w(v)/W(v) \mid \in N(S)$ }.
9: end if
10: $S \leftarrow S \cup \{v_{best}\}$.
12: end While
13: output: a complete solution S

5.Modified Carousel Greedy algorithm for the MWDS problem

CG algorithm requires two parameters, α and β be specified, where α is an integer and β is a percentage.

(i) As α increases from 1, running time increases and so does solution quality, until it stabilizes;

(ii) The parameter is related primarily to solution quality.

This procedure construct a solution S by $GreedyMWDS$ ordered by sequence of selection, the last selected elements are in the head, R is a partial solution produced by removing from head of S, β |S| elements, At each iteration (α |S|) we remove from tail of R one or more elements until the objective function value associated with the partial solution R is $\leq \gamma$ then using $GreedyMWDS$ to add an element to head of R .

using $GreedyMWDS$, add elements to R to obtain a feasible solution.

Algorithm 3 Procedure CG(α;β)

1: Input : $GreedyMWDS$
2: Let S the solution produced by $GreedyMWDS$
(ordered by sequence of selection, the last selected elements are in the head)
3: R the partial solution produced by removing from head of S, β |S| elements
4: Let the objective function value associated with the partial solution R
5: for α |S| iterations
6: remove from tail of R one or more elements until
the objective function value associated with the partial solution R is $\leq \gamma$
7: using $GreedyMWDS$, add an element to head of R
8: end for
9: using $GreedyMWDS$, add elements to R to obtain a feasible solution
10 : reduce(S, β)//see algorithm 4
11: output: a complete solution R

6. Elimination of redundant vertices

Solution S may contain redundant components (redundant vertices). A vertex $v \in S$ is redundant if all vertices from its closed neighborhood $N[v]$ are dominated by other vertices from S. Remember that $N[v] = N(v) \cup \{v\}$, where $N(v) = \{u \in V | (u,v)\ u \in S\}$ is the set of the neighbors of v that is, the set of neighbours of v

In this context, we assocoaite a a value reference(v) at each vetex v∈V defined as follow:

$$reference(v) = \begin{cases} |\{(u,v) \in V |\ u \in S\}|, v \notin S \\ |1 + \{(u,v) \in V |\ u \in S\}|, v \in S \end{cases}$$

It is clear that when S=∅ all vertices in V have white color and its reference value is equal to 0. Besides, after each adition of vertex v $\in V/S$ to the current solution its reference and the refrences of its neighbors are increased by one. Algoritm 4 provides the procedure reduce(S, β) that eliminates redunandant vertices.

Algorithm 4 Procedure reduce(S, β)

1: Input : a complete solution S and a parameter β
2: S^r = {u∈S | reference(v)>1: ∀v ∈ N[u]}
3: $\beta_0 \leftarrow$ random number uniformly distributed over [0,1].
3: while $S^r \neq \emptyset$ do
4: if $\beta \geq \beta_0$ then
5: $v^r \leftarrow$ argmin $\{d_S(v) / w(v) | v \in S^r)\}$.
6: else
7: $v^r \leftarrow$ argmin $\{ W(v)/w(v) | \in S^r \}$.
8: end if
9: $S \leftarrow S/\{v_r\}$
10: for each u∈ N[v] do
11: reference(u) ← reference(u) -1
12: end for
13: S^r = {u∈S | reference(v)>1: ∀v ∈ N[u]}
14: end while

7. Local Search for the MWDSP

Local search algorithms perform the search on problem's corresponding search space. The search space is implicitly defined by the way that the algorithm transforms a candidate solution into another. For the MWDS problem, local search algorithms usually maintain a candidate solution $S \subseteq V$ during the search. A vertex v is covered by S *if* $v \in N[S]$, and is uncovered otherwise. Also, the state of a vertex v is denoted by $s_v \in \{0,1\}$, such that s_v =1 means a vertex v is covered by a candidate solution S, and s_v =0 means it is uncovered. For a vertex, its age is defined as the number of steps since the last time it changed its state (being added or removed w.r.t. the maintained candidate solution S), and when we say the oldest vertex, we refer to the one with the minimum age value.

7.1. Configuration Checking

Local search suffers from the cycling phenomenon, i.e., revisiting a candidate solution that has been recently visited. This phenomenon wastes much computation time of a local search algorithm and more importantly prevents it from escaping from local optima.

To overcome the cycling problem, a number of methods have been proposed. The random walk strategy (Selman, Kautz, & Cohen, 1994) picks the solution component randomly with a certain probability and greedily makes the best possible move with another

probability. Random restarting (Houck, Joines, & Kay, 1996) is used to restart the search from another starting point of search space. Also, allowing non-improving moves with a probability, as in the Simulating Annealing algorithm, can provide more diversification to the search. Glover proposes the tabu method (Glover, 1989), which has been widely used in local search algorithms (Di Gaspero & Schaerf, 2007; Escobar, Linfati, Toth, & Baldoquin, 2014; Ahonen, de Alvarenga, & Amaral, 2014). To prevent the local search to immediately return to a previously visited candidate solution, the tabu method forbids reversing the recent changes, where the strength of forbidding is controlled by a parameter called tabu tenure. Besides these general methods dealing with the cycling phenomenon, there are also heuristics specialized for problems, such as the promising decreasing variable exploitation for the Boolean Satisfiability (SAT) problem (Li & Huang, 2005).

Recently, an interesting strategy called Configuration Checking (CC) (Cai et al., 2011) was proposed to handle the cycling problem in local search. Also, CC does not have instance-dependent parameters and is easy to use. The relationship between the tabu and CC strategy has been thoroughly discussed (Cai & Su, 2013). If the tabu tenure is set to 1, it can be proved that given a variable, if it is forbidden to pick by the tabu method, it is also forbidden by the CC strategy, while its reverse is not necessarily true.

The MWDS problem is in some sense similar to the vertex cover problem in that their tasks are both to find a set of vertices. Thus, we can easily devise a CC strategy for the MWDS problem, following the one for the vertex cover problem (Cai et al., 2011). An important concept of the CC strategy is the configuration of vertices. Typically, the configuration of a vertex v refers to a vector consisting of the states of all $v's$ neighboring vertices. The CC strategy for the MWDS problem can be described as following: given the candidate solution$S$, for a vertex $v \notin S$, if its configuration has not changed since $v's$ last removal from$S$, which means the circumstance of v has not changed, then v should not be added back to S.

An implementation of the CC strategy is to apply a Boolean array $confchange$ for vertices, where $confchange(v)$ =1 means that v is allowed to be added to S, and $confchange(v)$=0 on the contrary. In the beginning, for each vertex v, the value of $confchange(v)$ is initialized as 1; afterwards, when removing vertex x, $confchange(x)$ is set to 0; whenever a vertex v changes its state, for each vertex $u \in N(v)$, $confchange(u)$ is set to 1.

7.2. Two-Level Configuration Checking

Since the CC strategy has been successfully applied to solve several NP-hard combinatorial optimization problems, a natural question arises whether this strategy can also be applied to MWDS. Unfortunately, a direct application of the original CC strategy in local search for the MWDS problem does not result in an effective algorithm, and has poor performance on a large portion of the benchmark instances.

The original CC strategy would mislead the search by forbidding too many candidate vertices. In CC strategy, only the first-level neighborhood of a vertex is considered to avoid cycling, and the configuration of a vertex is considered changed only if at least one of its neighboring vertices changed its state. However, some analysis suggests that not only the first-level neighborhood but the second-level neighborhood are related to the cycling phenomenon and should be considered in the configuration of a vertex.

Inspired by this consideration, (Wang & Cai & Yin, 2017) [16] propose a new variant of CC for MWDS, which is referred to as two-level configuration checking (CC^2 for short), by redefining the configuration of vertices. In the CC^2strategy, we consider not only the first-level neighborhood (N_1) but also the second-level neighborhood (N_2).

7.2.1. Definition and Implementation of the CC2 Strategy

In this subsection, we define the CC^2 strategy and present an implementation for it. We start from the formal definition of the configuration of a vertex v.

Definition 1 Given an undirected graph $G = (V; E)$ and S the candidate solution, the configuration of a vertex $v \in V$ is a vector consisting of state of all vertices in $N_2(v)$.

Based on the above definition, we can define an important vertex in local search as follows.

Definition 2 Given an undirected graph $G = (V; E)$ and S the candidate solution, for a vertex $v \notin S$, v is configuration changed if at least one vertex in $N_2(v)$. has changed its state since the last time v is removed from S.

In the CC^2 strategy, only the configuration changed vertices are allowed to be added to the candidate solution S.

We implement CC^2 with a Boolean array $ConfChange$ whose size equals the number of vertices in the input graph. For a vertex v, the value of $ConfChange[v]$ is an indicator $ConfChange[v] = 1$ means v is a configuration changed vertex and is allowed to be added to the candidate solution S; otherwise, $ConfChange[v]$=0 and it cannot be added to S. During the search procedure, the $ConfChange$ array is maintained as follows.

CC2-RULE1. At the start of search process, for each vertex v, $ConfChange[v]$ is initialized as 1.

CC2-RULE2. When removing a vertex v from the candidate solution$S$, $ConfChange[v]$ is set to 0, and for each vertex $u \in N_2(v)$., $ConfChange[u]$ is set to 1.

CC2-RULE3. When adding a vertex v into the candidate solution S, for each vertex $u \in N_2(v).ConfChange[u]$ is set to 1.

To understand RULE2 and RULE3, we note that if $u \in N_2(v)$, then $\in N_2(u)$. Thus, if a vertex v changes its state (i.e., either being removed or added w.r.t. the candidate solution), the Configuration of any vertex $u \in N_2(v)$ is changed.

7.2.2. The Frequency based Scoring Function

(Wang & Cai & Yin, 2017) [16] introduce new scoring function The Frequency based Scoring Function

(Wang & Cai & Yin ,2017) [16] introduce a novel scoring function by taking into account of the vertices' frequency, which can be viewed as some kind of dynamic information indicating the accumulative effectiveness that the search has on the vertex. Intuitively, if a vertex is usually uncovered, then we should encourage the algorithm to select a vertex to make it covered.

In detail, in a graph, each vertex $v \in V$ has an additional property, frequency, denoted by $freq[v]$. the $freq$ of each vertex is initialized to 1. After each iteration of local search, the $freq$ value of each uncovered vertex is increased by one. During the search process, we apply the $freq$ of vertex to decide which vertex to be added or removed. Based on this consideration, we propose a new score function, which is formally defined as below.

Definition 3 For a graph $G = (V; E)$, and a candidate solution S, the frequency based scoring function denoted by $scoref$, is a function such that

$$freq[u] = \begin{cases} \frac{1}{\omega(u)} \times \sum_{u \in C_1} freq[v], u \notin S \\ -\frac{1}{\omega(u)} \times \sum_{u \in C_2} freq[v], u \in S \end{cases} \quad (5)$$

Where C_1=$N[u] \setminus N[S]$ and C2=$N[u] \setminus N[S \setminus \{u\}]$.

7. 3. The Selection Vertex Strategy

During the search process, for preventing visiting previous candidate solutions, we not only use the CC^2 strategy in the adding process, but also use the forbidding list in the removing process (Wang & Cai & Yin ,2017) [16] . The $forbid\_list$ used here is a tabu list which keeps track of the vertices added in the last step, and these vertices are prevented from being removed within the tabu tenure. In this sense, this frequency based prohibition mechanism can be viewed as an instantiation of the longer term memory tabu search, and the main difference is that our method also consider the information from the CC^2 strategy.

The algorithm picks a vertex to add or remove, using the frequency based scoring function and the above two strategies. Firstly, we give two rules for removing vertices.

REMOVE-RULE1. Removing one vertex v, which has the highest value of $scoref\ (v)$, breaking ties by selecting the oldest one.

REMOVE-RULE2. Removing one vertex v, which is not in $forbid\_list$ and has the highest value of $scoref\ (v)$, breaking ties by selecting the oldest one.

When the algorithm finds a solution, it removes one vertex from the solution and continues to search for a solution with smaller weight. In this process, we use REMOVE-RULE1 to pick the vertex. During the search for a solution, the algorithm exchanges some vertices, i.e., removing one vertex from the candidate solution and then iteratively adding vertices into the candidate solution. In this case, we select one vertex to remove according to REMOVE-RULE2. The rule to select the adding vertices is given below.

ADD-RULE. Adding one vertex v with $ConfChange[v] \neq 0$, which has the greatest value $scoref\ (v)$, breaking ties by selecting the oldest one. When adding one vertex into the candidate solution, we try to make the resulting candidate solution's cost (i.e., the total weight of uncovered vertices) as small as possible. When adding one configuration changed vertex with the highest value $scoref\ (v)$, breaking ties by preferring the oldest vertex.

7.4. CC^2FS Algorithm

Based on CC^2 and the frequency based scoring function, we develop a local search algorithm named CC^2FS. During the process of local search, we maintain a set from which the vertex to be added is chosen. The set for finding a vertex to be removed from the candidate solution is simply S.

$$CCV\ 2\ =\ \{v|ConfChange[v]\ =\ 1, v \notin S\}$$

The pseudo code of CC^2FS is shown in Algorithm 4. At first, CC2FS initializes $ConfChange$, $forbid\_list$ and the frequency and $scoref$ of vertices. Then it gets an initial candidate solution S greedily by iteratively adding the vertex that covers the most remaining uncovered vertices until S covers all vertices. At the end of initialization, the best solution $S*$ is updated by S.

After initialization, the main loop from lines 3 to 16 begins by checking whether S is a solution (i.e., covers all vertices). When the algorithm finds a better solution, $S*$ is updated. Then one vertex with the highest $scoref$ value in S is selected to be removed, breaking tie in favor of the oldest one. Finally, the values of $ConfChange$ are updated by CC2-RULE2. If there are uncovered vertices, CC^2FS first picks one vertex to remove from S with the highest value , breaking tie in favor of the oldest one. Note that when choosing a vertex to remove, we do not consider those vertices in $forbid\_list$, as they are forbidden to be removed by the forbidden list. After removing a vertex, CC2FS updates the $ConfChange$ values according to **CC2-RULE2**, and clear $forbid\_list$. Additional, since the tabu tenure is set to be 1, the $forbid\_list$ shall be cleared to allow previous forbidden vertices to be added in subsequent loop. After the removing process, CC^2FS iteratively adds one vertex into S until it covers all vertices, i.e. the candidate solution is a dominating set. CC^2FS first selects $v \in CCV2$ with the greatest $scoref(v)$, breaking ties in favor of the oldest one. When the picked uncovered vertex is added into the candidate solution, the $ConfChange$ values are updated according to CC2-RULE3 and this added vertex is added into the $forbid\_list$. After adding an uncovered vertex each time, the frequency of uncovered vertices is increased by one. When the time limit reaches, the best solution will be returned.

Algorithm 5: $CC2FS(G)$

Input: a weighted graph $G = (V; E; W)$
Output: dominating set of G

1: initialize $ConfChange$, $forbid\_list$, and the $freq$ and $scoref$ of vertices;
2: S Complete solution and $S* := S$;
3: if there are no uncovered vertices then
4: if $w(S) < w(S*)$ then $S* := S$;
5: v := a vertex in S with the highest value $scoref(v)$
6: $S := S \setminus \{v\}$ and update $ConfChange$ according to CC2-RULE2;
7: continue;
8: v := a vertex in S with the highest value $scoref(v)$ and v $\notin forbid\_list$
9: S := S $\setminus \{v\}$ and update $ConfChange$ according to CC2-RULE2;

10: $forbid\_list$:= ∅;
11: while there are uncovered vertices do
12: v := a vertex in CCV2 with the highest value $scoref(v)$
13: S := S ∪ $\{v\}$ and update $ConfChange$ according to CC2-RULE3;
14: $forbid\_list$:= $forbid\_list$ ∪ $\{v\}$;
15: $freq[v]$:= $freq[v]$ + 1, for $v \notin N[S]$;
16: return $S*$;

8. CG& CC2FS pseudo code for MWDSP

Algorithm 5: CG&CC2FS (G, cutoff)

Input: a weighted graph G = (V;E;W), cutoff time
Output: dominating set of G
S empty solution
1: while elapsed time < cutoff do
2: $GreedyMWDS$**(S;β) //see algorithm 2**
3: CG(α**;β) //see algorithm 3**
4: CC2FS (G) //see algorithm 5
5: If(S*<S) then (S*=S)
6: End while
7: return S*;

CHAPTER 4
ANALYSE OF RESULTS

Contents

1. Introduction……...36
2. Experimental evaluation……...……...36
   2.1. problem instances……..36
   2.2. Numerical results……...37

1. Introduction

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

2. Experimental evaluation

We have implemented this approach with c++ language using a bunch of known libraries as lists library and math.h library, and we have tested it on Windows 7 operating system with Intel(R) Celeron(R) CPU 13820 @ 1.70GHz 130 GHz and 4,00 Gega-bytes of memory CG& CC2FS was tested on a benchmark set of 530 problem instances. Its performance was compared with the one of recent metaheuristic
approaches available in the literature including the ACO approach (referred to as Raka-ACO) from [22]

2.1. Problem Instances

The set of benchmark instances was originally proposed in [8]. Each instance consists of an undirected, vertex weighted graph with n vertices and m edges. These instances are grouped with respect to their number of vertices n into two different classes:

a. **Class SMPI:** the class of small and medium problem instances (SMPI) contains 320 instances where n takes values from {50; 100; 150; 200; 250}.
b. **Class LPI:** the class of large problem instances (LPI) consists of 210 instances where n takes values from {300; 500; 800; 1000}.

The instances of the two classes share the following characteristics: (i) 10 problem instances were randomly generated per combination of n and m and results are presented as an average over the objective function values obtained for the 10 instances. (ii) The weight ω(v) of each vertex $v \in V$ is randomly drawn with respect to a uniform distribution from the interval [20; 70].for T1 benchmark

2.2. Numerical Results

The parameters involved in R-PBIG were set based on tuning by hand.
The following parameter values were adopted for the final experimental evaluation:
($\alpha = 2\ \beta = 40$) if the number of vertices less then 200 ($\alpha = 1\ \beta = 70$) otherwise
α : number increase the number of iteration
β : a percentage of destruction
(Cutoff =50 second) if the number of vertices less then 500(Cutoff =50 second) otherwise .Cutoff :Running time

Tables I and II summarize the performance comparison of CG& CC2FS with Raka-ACO on the Type I instances of classes SMPI and LPI respectively. The best obtained results are highlighted in boldface. The tables are organized as follows. The first columns define the instance size, in terms of the number of vertices (v) and the number of edges (e). Remember that for each combination of v and e (each table row) there are 10 different problem instances of the same size and the values shown in the tables represent the average results obtained for the 10 instances. Raka-ACO was not applied to all problem instances (as indicated by NA in someof the table columns).

Table 2: Comparison of results for SMPI for Type1 problems

Instance T1	Raka-ACO[22] MEAN	CG& CC2FS MEAN	RTime
v50e50	539.8	**533.700**	**7.145**
v50e100	391.9	**372.800**	**9.860**
v50e250	195.3	**.176700**	**6.513**
v50e500	112.8	**95.500**	**1.927**
v50e750	69.0	**63.500**	**4.067**
v50e1000	44.7	**41.5**	**0.905**
v100e100	1087.2	**1074.8**	**18.28**
v100e250	698.7	**630.100**	**16.750**
v100e500	442.8	**363.300**	**5.169**
v100e750	313.7	**265.700**	**15.047**
v100e1000	247.8	**215.100**	**10.685**
v100e2000	125.9	**108.300**	**7.369**
v150e150	1630.1	**1609.400**	**27.831**
v150e250	1317.7	**1252.900**	**24.702**
v150e500	899.9	**784.000**	**24.060**
v150e750	674.4	**584.300**	**24.208**
v150e1000	540.7	**451.600**	**18.807**
v150e2000	293.1	**257.800**	**13.677**
v150e3000	204.7	**169.400**	**12.946**
v200e250	2039.2	**1953.500**	**32.273**
v200e500	1389.4	**1283.200**	**27.789**
v200e750	1096.2	**952.800**	**29.378**
v200e1000	869.9	**764.700**	**21.976**
v200e2000	524.1	**436.900**	**24.483**
v200e3000	385.7	**309.200**	**19.238**
v250e250	NA	**1902.900**	**39.041**
v250e500	NA	**1442.000**	**28.632**
v250e750	NA	**1155.600**	**16.781**
v250e1000	NA	**663.800**	**24.707**
v250e2000	NA	**485.900**	**29.508**
v250e3000	NA	**478.200**	**23.709**
v250e5000	NA	**307.900**	**28.559**

Table 1 shows the numerical results for the instances of class SMPI (Type I). In 32 cases CG& CC2FS obtains the best result of the comparison.

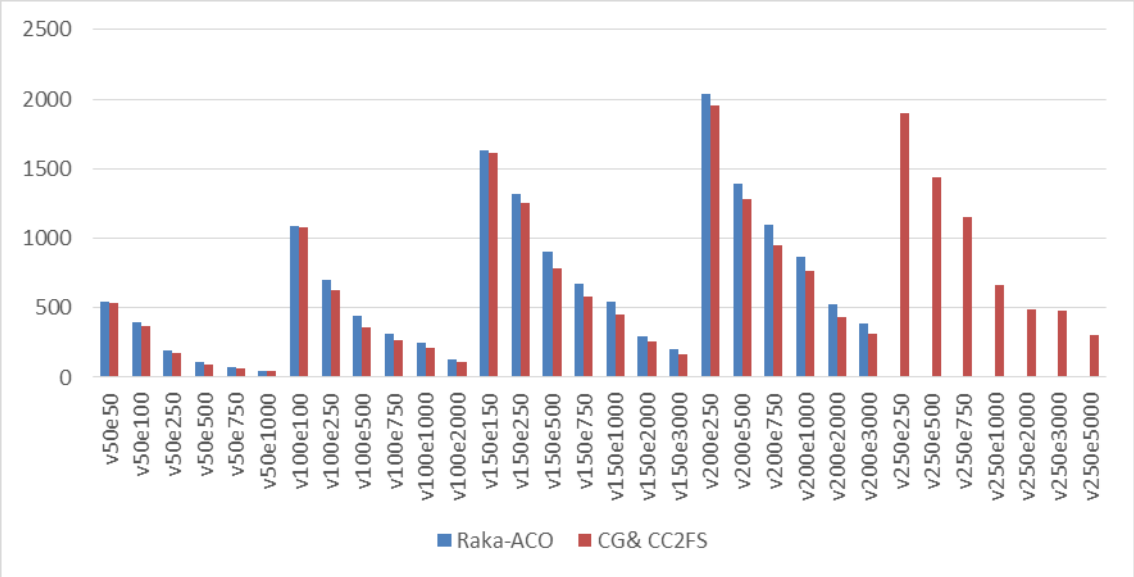


Figure 4.1**:** comparison with Raka-ACO on SMPI instances

Table 2: Comparison of results for large problems for Type1 problems

Instance T1	Raka-ACO[22] MEAN	CG& CC2FS MEAN	RTime
v300e300	NA	**3281.100**	**39.937**
v300e500	NA	**2539.500**	**41.678**
v300e750	NA	**1946.900**	**42. 490**
v300e1000	NA	**1596.800**	**33.863**
v300e2000	NA	**922.900**	**29.536**
v300e3000	NA	**672.300**	**25.618**
v300e5000	NA	**442.900**	**24.107**
v500e500	5476.3	5560.100	70.874
v500e1000	4069.8	**3828.400**	**83.014**
v500e2000	2627.5	**2360.100**	**58.453**
v500e5000	1398.5	**1140.000**	**51.081**
v500e10000	825.7	**640.1**	**45.165**
v800e1000	8098.9	8186.000	77.581
v800e2000	5739.9	**5427.100**	**75.396**
v800e5000	3116.5	**2678.900**	**76.025**
v800e10000	1923.0	**1560.200**	**73.038**
v1000e1000	10924.4	11319.100	42.098
v1000e5000	4662.7	**4046.700**	**67.596**
v1000e10000	2890.3	**2339.200**	**61.959**
v1000e15000	2164.3	**1686.100**	**303.306**
v1000e20000	1734.3	**1322.700**	**84.647**

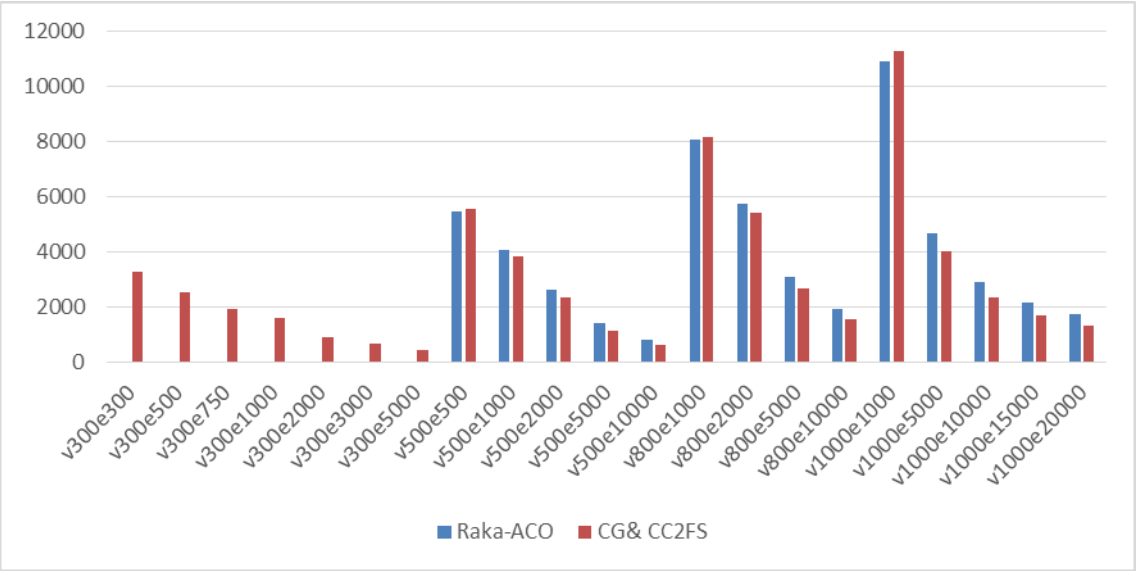


Figure 4.2**:** comparison with Raka-ACO on LPI instances

The results for the instances of class LPI (Type I) are presented in Table 2.On the LPI instances we can notice also that there is some instance the Raka-ACO algorithm can't calculate with other instances it's better than our algorithms but in the majority of instances our algorithm is better than Raka-ACO

Conclusion and future works

Minimum weight dominating set (MWDS) finds many uses in solving problems as varied as clustering in wireless networks, multi-document summarization in information retrieval and so on. It is proven to be NP-hard.

The proposed approach for solving the MWDSP method is a hybridization of modified Carousel Greedy algorithm and Local search the Carousel greedy algorithm based on Two principal operations destruct and construct a solution until find a stabilized solution and local search based on a new configuration checking strategy namely CC^2 based on the two-level neighborhood of vertices to remember the relevant information of removed and added vertices and prevent visiting the recent paths. Moreover, (Wang & Cai & Yin ,2017) [16] introduce a new frequency based scoring function for solving MWDS. The experimental results showed that CG&CC2FS performs essentially better than state of the art algorithms on almost all instances in terms of solution quality

As for future work, we consider to further improving the CG&CC2FS algorithm by integrating some other ideas Also we would like to test our algorithms on other instances including larger graphs and apply it on another problems.

Bibliography

Books

[1] A. Astolf, Optimization An Introduction, September 2006.

[2] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, Introduction to algorithms, MIT Press, Massachusetts, USA, 3 edition, 2009.

[3] D.Guichard, An Introduction to Combinatorics and Graph Theory.

[4] A,R, Sarker and S.N.Charles, Optimization Modelling A ractical Approach, Taylor and Francis Group , 2008.

[5] A. Tang, Graph Theory, IMO Training, 2008

Papers

[6] S.Bouamama, C.Blum, A randomized population-based iterated greedy algorithm for the minimum weight dominating set problem, 2015

[7] M. Bouchakour,T.M. Contenzab, C.W. Leec, A.R. Mahjoub,On the dominating set polytope,Elsivier , vol 29, 2007 .

[8] C.Cerrone, R.Cerulli and B.Golden, Carousel Greedy: A Generalized Greedy Algorithm with Applications in Optimization, Computers and Operations Research,2017.

[9] V. Chvatal, A greedy heuristic for the set-covering problem, Mathematics of operations research, vol 4,1979.

[10] S.Desale, A.Rasool, S.Andhale and P.Ranem, Heuristic and Meta-Heuristic Algorithms and Their Relevance to the Real World: A Survey.international journal of computer engineering in reserch trends, vol 2, 2015.

[11] H. H. Hoos and T. Stiitzle, Stochastic local search Foundations and applications. Elsevier, 2004.

[12] R. MartÏ and G. Reinelt, The Linear Ordering Problem Exact and Heuristic Methods in Combinatorial Optimization, Springer, 2011.

[13] A.Potluri and A. Singh, Hybrid metaheuristic algorithms for minimum weight dominating set, Applied Soft Computing,vol 13 , 2013.

[14] C.C. Ribeiro, G.C.Mauricio, Resende, Optimization by GRASP Greedy Randomized Adaptive Search Procedures, 2016.

[15] F. Rothlauf, Design of Modern Heuristics, Natural Computing Series, 2011.

[16]Y.Wang, S.Cai &M.Yin, Local Search for MinimumWeight Dominating Set with Two-Level Configuration Checking and Frequency Based Scoring Function, Journal of Artificial Intelligence Research, vol 58,2017.

Thesis

[17] X. Yang. Engineering optimization An introduction with metaheuristic applications, Jhon Wiley and Sons, Inc, Hoboken, New Jersey,2010.

[18] S.Bouamama, Design of a Learning Method for Automatic Data Extraction, PhD thesis, Setif 2013.

[19] M.Iori, Metaheuristic Algorithms for Combinatorial Optimization Problems, PhD thesis, Bologna, 2003.

Websites

[20] J. W. Chinneck. Practical optimization: A gentle introduc-tion. 2012.
http://www.sce.carleton.ca/faculty/chinneck/po.html visited 06/05/2017 17:50

[21]https://optimization.mccormick.northwestern.edu/index.php/Computational_complexity visited 08/05/2017 21:50

[22] R. Jovanovic, M. Tuba, D. Simian, Ant colony optimization applied to minimum weight dominating set problem, in: Proceedings of the 12th WSEAS International

[23] Conference on Automatic Control, Modelling & Simulation, 2010, pp. 322–326.

M. Garey, D.S. Johnson, Computers and Tractability, A Guide to the Theory of NP-Completeness, Freeman and Company, New York, 1979.

ملخص

نعالج في هدا العمل أحد المشاكل نظرية المخططات و المعروفة بتطبيقات مهمة:
مشكلة المجموعة المهيمنة ذات الوزن الادنى والتي تعرف ضمن المسائل كثيرة الحدود (NP-complete problems) غير القطعية الكاملة.
حيث نقوم باقتراح خوارزميتة هجينة تجمع بين طريقة البحث المحلي و خوارزمية جشعة معدلة لايجاد حلول مقبولة في وقت حساب معقول لهذه المشكلة. وتظهر النتائج التجريبية أن الخوارزمية المقترحة لها أداء تنافسي مع خوارزمبة مملكة النمل المنشورة مؤخرا.
الكلمات المفتاحية خوارزمية جشعة, البحث المحلي, المجموعة المهيمنة ذات الوزن الادنى

Abstract

In this memory, we deal with a classical problem in graph theory the minimum weight dominating set problem. The latter belong to the class of NP-complete problems where no efficient algorithm is known to solve it to optimality. We have implemented a hybrid algorithm that combines a modified carousel greedy algorithm and local search to give near optimal solutions within a reasonable computation-time to this problem. Experimental results show that the algorithm has competitive performance with a recent published ant colony optimization approach.

Keywords: NP-Complete,Carousel Greedy,Local Search,minimum weight dominating set problem

Résumé

Dans ce mémoire, nous abordons un problème classique dans la théorie des graphes, le problème de l'ensemble dominant de poids minimum. Ce dernier appartient à la classe des problèmes NP-complets et pour lequel il n'existe pas actuellement un algorithme efficace capable de le résoudre à l'optimalité. Nous avons mis en place un algorithme hybride qui combine un algorithme glouton – carrousel et une approche de la recherche locale pour obtenir des bonnes solutions en un temps de calcul raisonnable pour ce problème. Les résultats expérimentaux montrent que l'algorithme propose à une performance concurrentielle avec une métaheuristique récente basée sur les algorithmes des colonies des fourmis.

Mots Clés:NP-Complet,Carousel Glouton,Recherche local,l'ensemble dominant de poids minimum