

République Algérienne Démocratique et Populaire
Ministère de l'Enseignement Supérieur et de la Recherche Scientifique
Université Ferhat Abbas de Sétif –1–



THÈSE

Présentée à la Faculté des Sciences
Département d'informatique
En vue de l'obtention du diplôme de

Doctorat en Sciences

Option : INFORMATIQUE

Par

Salim BOUAMAMA

Thème

Design of a Learning Method for Automatic Data Extraction

Soutenue le : / / 2013 devant le jury composé de

President:	M. Tayeb LASKRI	Prof.	Université d'Annaba
Examineur:	Okba KEZZAR	Prof.	Université de Biskra
	Mustapha BOURAHLA	MCA	Université de M'sila
	Nadjet KAMEL	MCA	Université de Sétif 1
Rapporteur:	Abdallah BOUKERRAM	Prof.	Université de Sétif 1
Co-rapporteur:	Christian BLUM	Prof.	IKERBASQUE/UPV Donostia Espagne
Invité:	Allaoua REFFOUFI	MCA	Université de Sétif 1

Contents

1	Introduction	1
1.1	Overview	1
1.2	Our contributions	5
1.3	Thesis organization	8
1.4	Academic Publications Produced	9
2	Background	11
2.1	Definition of an optimization problem	11
2.2	Classification of optimization problems	12
2.3	Computational complexity	15
2.3.1	Definitions and basic concepts	16
2.3.2	Complexity classes	18
2.3.2.1	Class \mathcal{P}	19
2.3.2.2	Class \mathcal{NP}	19
2.3.2.3	Class \mathcal{NP} -complete	19
2.3.2.4	Class \mathcal{NP} -hard	20
2.4	Local Versus Global Optima	21
2.5	Optimization methods	22
2.5.1	Exact methods	23
2.5.2	Approximate methods	24
2.5.2.1	Heuristics	25
2.5.2.2	Metaheuristics	26
2.6	Classification of metaheuristics	28
3	Stochastic optimization techniques	31
3.1	Search techniques	32

3.1.1	Blind search strategies	32
3.1.2	Heuristic search strategies	32
3.2	Iterated local search	33
3.3	Greedy Randomized Adaptive Search Procedure	36
3.4	Genetic algorithm	38
3.5	Ant colony Optimization	40
3.5.1	Swarm intelligence	40
3.5.2	The Basic Principle of Ant Colony Optimization	41
3.5.3	Solution construction	43
3.5.4	local improvement	44
3.5.5	Pheromone update	44
3.6	Motif finding using ant colony optimization	45
3.6.1	The Motif Finding Problem	47
3.6.2	Basic Motif Representations	48
3.6.2.1	Profile Model	48
3.6.2.2	consensus Model	50
3.6.3	The Basic Gibbs Sampling Algorithm	51
3.6.4	The Proposed Approach (MFACO)	53
3.6.4.1	Initialization	54
3.6.4.2	Solution Construction	54
3.6.4.3	Pheromone Update	56
3.6.5	Computational Experiments	57
3.6.6	Concluding remarks	60
4	Minimum Weight Vertex Cover Problem	67
4.1	Preliminaries on graphs	68
4.1.1	Graphs	68
4.1.2	The degree of a vertex	69
4.1.3	Graph representations	69

4.1.3.1	Adjacency list representation	69
4.1.3.2	Adjacency matrix representation	70
4.2	Minimum Weight Vertex Cover Problem	71
4.2.1	Problem statement	71
4.2.2	Example	72
4.3	MWVCP complexity	73
4.4	Approximation algorithms for MWVCP	73
4.4.1	Bar-Yehuda and Even's algorithm	73
4.4.2	Pitt's randomized algorithm	74
4.4.3	Clarkson's Greedy algorithm	74
5	Population Based Iterated Greedy Algorithm for MWVCP	76
5.1	Greedy paradigm	76
5.2	Iterated greedy search	79
5.3	Design of a PBIG algorithm for MWVCP	80
5.3.1	Population initialization	83
5.3.2	Construction phase	83
5.3.3	Destruction phase	85
5.3.4	Complexity Considerations	85
6	Experimental Results	87
6.1	Problem Instances	88
6.2	Algorithm Tuning	89
6.3	Results and discussions: PBIG versus ACO and ACO+SEE	91
6.3.1	Results Concerning Class SPI	91
6.3.2	Results Concerning Class MPI	94
6.3.3	Results Concerning Class LPI	95
6.3.4	Convergence speed of PBIG	96
6.4	Results and discussions: PBIG versus HSSGA	96

Contents	iv
6.4.1 Results Concerning Class SPI	97
6.4.2 Results Concerning Class MPI	97
6.4.3 Results Concerning Class LPI	98
7 Conclusions and Future Works	111
7.1 Conclusions	111
7.2 Future Works	113
A Best solutions of case $(n = 20, m = 80)$	115
Bibliography	117

List of Figures

1.1	The process of the optimization cycle.	2
2.1	Classification of optimization problems	14
2.2	Graphic example of the O notation.	17
2.3	The relationships among complexity classes of problem (Assuming $P \neq NP$).	21
2.4	Local Versus Global Optima	22
2.5	Classical optimization methods	24
3.1	The principle of ILS. Starting with a local minimum s , a perturbation is performed, leading to a solution s^p . After applying a local search procedure (LS), we may find a new local minimum s' that is better than s	35
3.2	An example illustrating the two models of motif representation.	49
3.3	Problem graph-based representation	55
4.1	Graph representations	70
4.2	An illustrative example of MWVCP	72
6.1	Convergence speed of ACO and ACO+SEED for the test case of three problem instances of class LPI [Jovanovic 2011].	103
6.2	Convergence speed for problem instance $n = 500, m = 2000$	104
6.3	Convergence speed for problem instance $n = 800, m = 10000$	104
6.4	Convergence speed for problem instance $n = 1000, m = 1000$	105

List of Tables

3.1	Dataset 1, $l = 7$	61
3.2	Dataset 1, $l = 13$	61
3.3	Dataset 2, $l = 7$	62
3.4	Dataset 2, $l = 13$	62
3.5	Dataset 3, $l = 7$	63
3.6	Dataset 3, $l = 13$	63
3.7	Performance results achieved by GS, BP, and MDGA in the CRP dataset [Che 2005].	64
3.8	Performance results achieved by MFACO in the CRP dataset.	65
3.9	The consensus sequences of motifs discovered by MFACO, foot- printing, Gibbs Sampler, BioProspector, and MDGA respec- tively. (based on the results from Table 3.7 and Table 3.8). . .	66
6.1	Performance of PBIG, ACO and ACO+SEED on instances of class SPI (Type I).	92
6.2	Performance of PBIG, ACO and ACO+SEED on instances of class SPI (Type II).	99
6.3	Performance of PBIG, ACO and ACO+SEED on instances of class MPI (Type I).	100
6.4	Performance of PBIG, ACO and ACO+SEED on instances of class MPI (Type II).	101
6.5	Performance of PBIG, ACO and ACO+SEED on instances class LPI.	102
6.6	Performance of PBIG and HSSGA on instances of class SPI (Type I).	106

6.7	Performance of PBIG and HSSGA on instances of class SPI (Type II).	107
6.8	Performance of PBIG and HSSGA on instances of class MPI (Type I).	108
6.9	Performance of PBIG and HSSGA on instances of class MPI (Type II).	109
6.10	Performance of PBIG and HSSGA on instances of class LPI. .	110

Acknowledgements

I would like to express my deep gratitude to my supervisor, Pr. Abdallah Boukerram, for his invaluable advice, time and support throughout this research work.

I would like to thank Pr. Mohamed Tayeb Laskri, Pr. Okba Kezzar, Dr. Mustapha Bourahla, Dr. Nadjat Kamel, and Dr. Allaoua Reffoufi for accepting to be members of the examination committee for this thesis and for taking time to read and review it. Their suggestions will be taken into account and will surely significantly improve the final version.

I am also very grateful to Pr. Christian Blum for hosting me and giving me the opportunity to complete my doctoral research under his supervision at the *Universitat Politècnica de Catalunya* (Barcelona), for making available all the resources needed to carry out this research, and for opening the doors to the various interesting topics treated in this work.

Finally, I must thank my family and dearest friends for their understanding and constant support over the years.

DESIGN OF A LEARNING METHOD FOR AUTOMATIC DATA EXTRACTION

Abstract

Optimization is a scientific discipline that is concerned with the extraction of optimal solutions for a problem, among alternatives. Many challenging applications in business, economics, and engineering can be formulated as optimization problems. However, they are often complex and difficult to solve by an exact method within a reasonable amount of time.

In this thesis we propose a novel approach called population based iterated greedy algorithm in order to efficiently explore and exploit the search space of one of the NP-hard combinatorial optimization problems namely the minimum weight vertex cover problem. It is a fundamental graph problem with many important real-life applications such as, for example, in wireless communication, circuit design and network flows.

An extensive experimental evaluation on a commonly used set of benchmark instances shows that our algorithm outperforms current state-of-the-art methods not only in solution quality but also in computation time.

Keywords: stochastic algorithms, combinatorial optimization problem, minimum weight vertex cover problem, greedy heuristic

CONCEPTION D'UNE METHODE AUTOMATIQUE D'APPRENTISSAGE POUR LA FOUILLE DE DONNEES

Résumé

L'optimisation est une discipline scientifique qui s'intéresse à l'extraction des solutions optimales pour un problème donné, présentant plusieurs solutions. De nombreuses applications stimulantes dans l'industrie, en finances et en ingénierie peuvent être formulées comme des problèmes d'optimisation. Cependant, ils sont souvent complexes et difficiles à résoudre avec exactitude dans des temps de calcul acceptables.

Dans cette thèse, nous proposons une nouvelle approche appelée algorithme glouton itératif à base de population pour explorer et exploiter d'une manière adéquate l'espace de recherche de l'un des problèmes d'optimisation combinatoires de classe NP-difficile. Il s'agit du problème de la couverture par les sommets de poids minimum qui est un problème fondamental dans la théorie de graphe, couvrant de nombreuses applications dans les domaines de la communication sans fils, de la conception des circuits et du contrôle du flux dans les réseaux.

Une évaluation expérimentale approfondie sur les différents jeux de données disponibles dans la littérature montre que l'algorithme développé, concurrence aisément en terme de qualité des solutions et des temps de calcul associés, les méthodes de pointe actuelles dans la plupart des cas.

Mots-clés: algorithmes stochastiques, optimisation combinatoire, problème de la couverture par les sommets de poids minimum, heuristique glouton.

Introduction

Contents

1.1 Overview	1
1.2 Our contributions	5
1.3 Thesis organization	8
1.4 Academic Publications Produced	9

1.1 Overview

Optimization is a scientific discipline that is concerned with the detection of optimal solutions for a problem, among available (finite or infinite number of) alternatives. The optimality of solutions is based on one or several criteria that are usually problem and user dependent [Parsopoulos 2010]. It may occur in the minimization of time, cost, and risk or the maximization of profit, quality, and efficiency [El-Ghazali 2009].

Optimization involves the study of optimality criteria for problems, the determination 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 supposed 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 [Chinneck 2012].

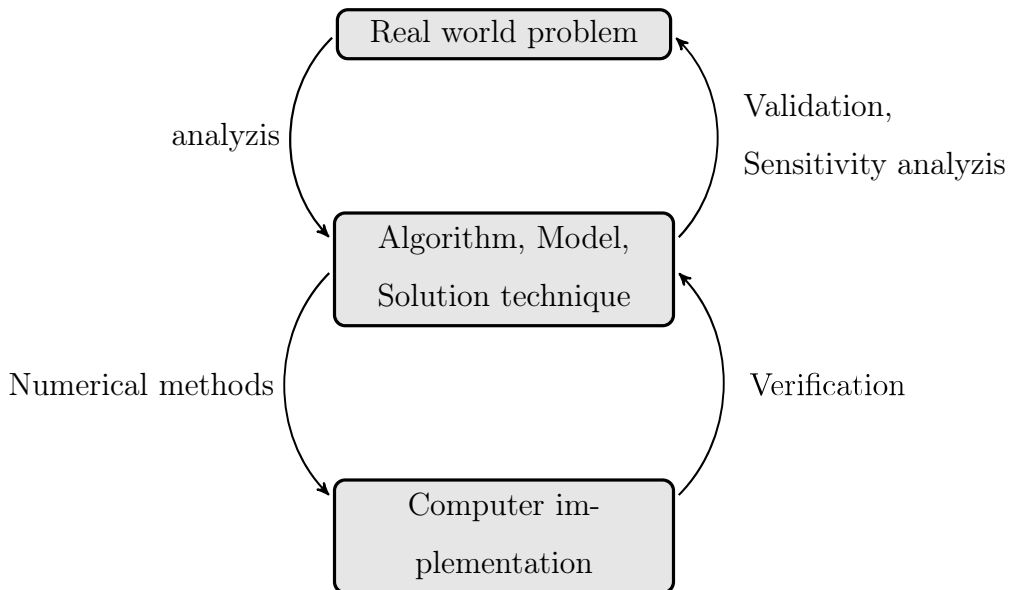


Figure 1.1: The process of the optimization cycle.

An optimization problem begins with a set of independent variables or parameters, and often includes conditions or restrictions that define the ac-

ceptable values of the variables. Such restrictions are termed the constraints of the problem. Another essential component is a measure of goodness called the objective function, which depends in some way on the variables. The solution of an optimization problem (not necessarily unique) is a set of allowed values of the variables for which the objective function achieves an optimal value (its maximum or minimum value)[Gill 1982]. Many challenging applications in business, economics, and engineering can be posed 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. Some of the reasons why this is so include [Spall 2003]:

- Presence of noise in solution evaluation as the objective function is not always either easily measurable or possible to write it in the explicit form.
- Difficulties in distinguishing global from local optimal solutions (The quality of solutions is not very attractive).
- Exponential growth of the search space with problem dimension (Excessive computation time).
- Difficulties associated with constraints and the need for problem-specific methods.
- The lack of stationarity in the solution as a result of the conditions of the problem changing over time.

This thesis describes an investigation into the development of a stochastic optimization approach to solve the minimum weight vertex cover problem (MWVCP) which is a fundamental graph problem with many important real-life applications such as, for example, in wireless communication, circuit design and network flows. Given an undirected graph $G = (V, E)$ with weights on

its vertices, the goal of MWVCP is to find among all subsets $S \subset V$ that are vertex covers a subset S^* for which the sum of the weights of the vertices is minimal. A vertex cover of a graph is a subset of vertices such that each edge has at least one endpoint in the subset. MWVCP is one of the classic Karp’s 21 NP-complete problems [Karp 1972] for which no polynomial-time algorithm is known to solve it at optimality. Due to its inherent complexity, classical approaches are not practical since the time involved to compute an optimal solution increases exponentially with the size of the problem making an exhaustive search impossible due to the combinatorial nature of the problem. As well as many other difficult optimization problems, one is forced to compromise. Thus, instead of looking for algorithms that provide optimal solutions in polynomial time, an efficient algorithm that can generate near-optimal solutions is more desirable. In the last decades, a variety of methods have been devised to deal with this problem and most of them are based on heuristics for providing approximate solutions in reasonable computation time. In fact, most of the developed algorithms are greedy heuristics. The first one was introduced in [Chvátal 1979]. It starts with an empty partial solution and adds one vertex at a time. More specifically, at each step it chooses the vertex with the minimum ratio between vertex weight and current degree. Note that the *current degree* is only determined by edges that are not yet covered. Pitt [Pitt 1985] proposed a randomized 2-approximation algorithm which repeatedly selects one end-point (vertex) of an arbitrarily chosen edge with a probability inversely proportional to its weight. From complexity theory it is known that—unless $P = NP$ —it is impossible to approximate the vertex cover problem, which is a special case of MWVCP in which all vertices are of unit weight, within any factor smaller than $10\sqrt{5} - 21 \simeq 1.3606$ [Dinur 2005]. Moreover, it might be hard to approximate it within an approximation ratio of $2 - \varepsilon$ for all $\varepsilon > 0$ [Khot 2003]. In addition to heuristics, several metaheuristics have been proposed for tackling MWVCP. Khuri and Bäck [Khuri 1994]

treated MWVCP as a constrained combinatorial optimization problem and introduced a genetic algorithm that makes use of a graded penalty term incorporated into the fitness function to penalize infeasible solutions. Furthermore, a steady-state genetic algorithm hybridized with a greedy heuristic has been reported in [Singh 2006]. The genetic algorithm generates vertex covers which are then refined by the greedy heuristic. That is, the heuristic first finds the set of those vertices in the vertex cover whose edges are fully covered by other vertices within the cover. Then it consecutively quasi-randomly removes the vertex, having the maximum ratio of weight to degree, from this set from the vertex cover until the set is empty. Recently, an ant colony optimization approach (ACO) was proposed by Shyu et al. in [Shyu 2004]. According to their empirical results ACO outperforms others metaheuristics such as tabu search and simulated annealing. This algorithm was further improved in [Jovanovic 2011] by incorporating a pheromone correction strategy in the hope to avoid stagnation of the search and the convergence to local optima. An optimization approach based on the law of gravity has been described in [Balachandar 2009]. The results of this approach were compared to genetic algorithms and some greedy heuristics. More recently, Voß and Fink [Voß 2012] propose to solve MWVCP by means of a hybridization of reactive tabu search and simulated annealing. However, these proposed approaches generally seem not to scale well. Moreover, they seem to be quite time consuming for large size problem instances.

1.2 Our contributions

This research has been motivated by an interest in developing a new variant of the iterated greedy algorithm capable of tackling MWVCP in a more effective way than currently exists. The proposed approach called population-based iterated greedy algorithm (PBIG) is able to provide state-of-the-art

results in a short computational time. Several studies from the literature have shown that constructive heuristics may be improved by a simple meta-heuristic framework known as an iterated greedy algorithm (IG). Our approach belongs to this class of algorithms having recently received considerable attention regarding their potential for effectively solving a wide range of difficult discrete/combinatorial optimization problems, including the founder sequence reconstruction problem [Benedettini 2010], various scheduling problems [Ruiz 2007, Fanjul-Peyro 2010, Ribas 2011] and the maximum diversity problem [Lozano 2011]. At each iteration, IG algorithms first partially destroy the incumbent solution before a greedy heuristic is used to derive a complete solution from the partially destroyed solution [Hoos 2004]. According to Hoos and Stützle [Hoos 2004], two important advantages of starting the solution construction from partial solutions are that (i) the solution construction process is much faster and (ii) good parts of solutions may be exploited directly. Our contributions can be summarized in the following points.

- A crucial issue in the design of an IG algorithm is the choice of the constructive greedy heuristic, which has a great impact on its performance. For the construction phase of PBIG, we develop an efficient randomized greedy construction heuristic based on the heuristic described in [Chvátal 1979, Clarkson 1983]. During the destruction phase, a fixed number of randomly chosen vertices (generally referred to as solution components) are removed from the incumbent solution. Hereby, the size of destruction is determined as a fixed proportion of the incumbent solution.
- We carry out extensive parameter tuning experiments for the proposed algorithm to investigate the effects of parameter tuning of performance using a cluster of PCs equipped with Intel Xeon X3350 processors with 2667 MHz and 8 Gigabytes of memory.

- We assess the performance of our algorithm on all benchmark instances that have been recently considered by many researchers, such as Shyu et al. [Shyu 2004], Sing and Gupta [Singh 2006], Balachandar and Kannan [Balachandar 2009], and more lately Jovanovic and Tuba [Jovanovic 2011]. In other words, PBIG was tested on 1125 graphs of different size, ranging from a few dozen to about 1000 nodes.
- With the aim of increasing the efficiency of the proposed algorithm, we extend it to be able to work with a population of solutions, rather than being restricted to a single incumbent solution. That is, we investigated how its performance is affected by the size of evaluated solutions for each PBIG iteration. The experimental evaluation shows that this extension contributes significantly to the quality of the obtained results.

the experimental results show that PBIG is able to outperform recent state-of-the-art algorithms such as the ones proposed in [Shyu 2004, Singh 2006, Balachandar 2009, Jovanovic 2011]. Another advantage of our approach is the fact that it is able to derive good solutions for difficult problem instances in a short computation time.

During the first phases of our research we have also implemented an ant colony optimization (ACO) [Dorigo 2004, Dorigo 1996] approach for the motif finding problem (See Section 3.6) as it is considered as one of the challenging problems in molecular biology, which aim to identify the specific binding sites of transcription factors in the promoter regions of genes. The motif finding problem belongs also to the class of NP-complete problems, and therefore the existence of a polynomial time algorithm to solve the problem optimally is unlikely. We design a hybrid approach combining $\mathcal{MAX-MIN}$ Ant System [Stützle 2000], one of the best performing variants of ACO metaheuristic, and a deterministic Gibbs sampling approach [Lawrence 1993], plays the role of a local heuristic optimization step, to give a near optimal solution to the

motif finding problem in a reasonable amount of time. Unlike other motif finding techniques, our approach called Motif Finding based on Ant Colony Optimization (MFACO) searches both in the space of starting positions as well as in the space of motif patterns. So that, it has more chances to find potential motif patterns. Although the method is also valid for protein sequences, we apply it only to DNA sequences.

To judge the goodness of MFACO, a series of computational experiments are performed on some datasets including the *Escherichia coli* CRP protein dataset. Its performance was compared with other recent proposed algorithms for finding motifs such as Genetic Algorithms [Liu 2004, Che 2005], MEME [LcBailey 1994], MotifSampler [Thijs 2001], and BioProspector [Liu 2001].

1.3 Thesis organization

The outline of the thesis is as follows. Chapter 2 provides the theoretical background of the thesis. It gives a brief review of both optimization problems and of the techniques used for solving such problem. It begins with a definition of optimization problems and introduces a possible classification of them. To help clarify the inherent difficulty of optimization problems, the key terms and notations in the context of problem complexity that pertain to our research are then presented. The remaining part of the chapter is devoted to a discussion of different types of problem solving techniques. It briefly discusses exact methods, heuristics, and metaheuristics.

The aim of chapter 3 is twofold. The first is to present some of well-known stochastic optimization techniques that are used for solving optimization problems especially those that deal with combinatorial optimization problems such as ant colony optimization, genetic algorithm, iterated local search, and greedy randomized adaptive search procedure. The second explains how ant colony optimization can be applied successfully to the motif finding problem. We

present the motif finding problem and explain in detail the main algorithmic components of the proposed approach to deal with this problem. A discussion of the experimental results is also provided.

Chapter 4 is devoted to the minimum weight vertex cover problem. First, it recalls some basic definitions of graph theory. Then, the problem is formally defined and illustrated by an example. After mentioning the complexity of MWVCP, the related works from the literature are reported. A particular attention is given to three approximation algorithms based on greedy heuristics.

Chapter 5 is the motivation of this research work. It describes in more detail the framework of the proposed approach (PBIG). In order to help understanding the basic structure of PBIG a pseudo-code is given for its main algorithmic components.

Chapter 6 reports the results of an extensive experimental performance evaluation of PBIG on several well-known benchmark datasets. We first describe the set of benchmark instances that have been used to test our approach. Then, we report on the tuning process that has been employed for finding well-working settings for the algorithm parameters. Finally, a comparison of the performance of PBIG against the performance of recently published algorithms is provided.

The last chapter concludes the thesis and highlights an outlook on the future work.

1.4 Academic Publications Produced

The following academic papers have been produced as a result of this research.

- Salim Bouamama, Christian Blum and Abdellah Boukerram. *A population based iterated greedy algorithm for the minimum weight vertex cover problem*. Applied Soft Computing, vol. 12, no. 6, pages 1632-1639. Else-

vier, 2012. <http://www.sciencedirect.com/science/article/pii/S1568494612000737>. ISSN: 1568-4946, Impact Factor: 2.612.

- Salim Bouamama, Christian Blum and Abdellah Boukerram. *A population based iterated greedy algorithm for the minimum weight vertex cover problem*. Accepted for oral presentation at the 21st International Symposium on Mathematical Programming (ISMP), Berlin, 2012.
- Salim Bouamama, Abdellah Boukerram and Amer F. Al-Badarneh. *Motif Finding Using Ant Colony Optimization*. In Marco Dorigo et al., editors, Swarm Intelligence, volume 6234 of *Lecture Notes in Computer Science*, pages 464-471. Springer Berlin Heidelberg, 2010. http://link.springer.com/chapter/10.1007/978-3-642-15461-4_45. ISBN: 978-3-642-15460-7.

Background

Contents

2.1	Definition of an optimization problem	11
2.2	Classification of optimization problems	12
2.3	Computational complexity	15
2.3.1	Definitions and basic concepts	16
2.3.2	Complexity classes	18
2.4	Local Versus Global Optima	21
2.5	Optimization methods	22
2.5.1	Exact methods	23
2.5.2	Approximate methods	24
2.6	Classification of metaheuristics	28

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

[Yang 2010]:

$$\underset{x \in \mathcal{S}}{\text{minimize}} \quad f_m(x), \quad (m = 1, \dots, M), \quad (2.1)$$

$$\text{subject to} \quad \phi_j(x) = 0, \quad (j = 0, \dots, J), \quad (2.2)$$

$$\psi_k(x) \leq 0, \quad (k = 0, \dots, K), \quad (2.3)$$

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 x_i 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 \mathcal{S} , 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*.

2.2 Classification of optimization problems

There are many possible ways of classifying optimization problems since it is not well established and there is some confusion in literature. Classification can be carried out with respect to (see Figure 2.1) [Yang 2010]:

- **The number of objectives:** in the case of $M = 1$ we have a *single objective optimization problem* and the other case, $M > 1$, corresponds to the *multiobjective optimization problem*. The latter is also referred to as multicriteria or even multi-attributes optimization in literature and most real-life problems are of this form of optimization.
- **The existence of constraints:** any optimization problem may be

classified as constrained or unconstrained, depending on whether or not constraints exist in the problem. *An unconstrained optimization problem* implies that there is no constraint at all $J = K = 0$. When $J + K > 0$ it becomes a *constraint optimization problem*.

- **The function form:** The objective functions can be either linear or nonlinear. If the constraints ϕ_j and ψ_k are all linear, in consequence a *linearly constrained problem* is obtained. On the other hand when all the functions of the design vector are nonlinear, the problem is called a *nonlinear optimization problem*.
- **The landscape of the objective functions:** for a single objective function, the shape or the landscape may vary significantly in the non-linear case. If there is only a single valley or peak with a unique global optimum, then the optimization problem is said to be *unimodal*. In this case, the local optimum is the only optimum, which is also its global optimum. However, most objective functions have more than one mode, and such multimodal functions are much more difficult to solve.
- **The type of the value of the design variables:** if the values of all design variables are discrete or can be reduced to discrete, then the optimization is called *discrete optimization* as opposed to continuous optimization where all the design variables are continuous or taking real values in some interval. Discrete optimization can be divided into two categories: *integer programming problem* where all variables are restricted to take integer values and *combinatorial optimization problem*. Combinatorial optimization involves the search for an optimal object in a finite collection of objects which has typically a concise representation such as a graph [Schrijver 2003].
- **The uncertainty in values:** So far, we only consider the case when

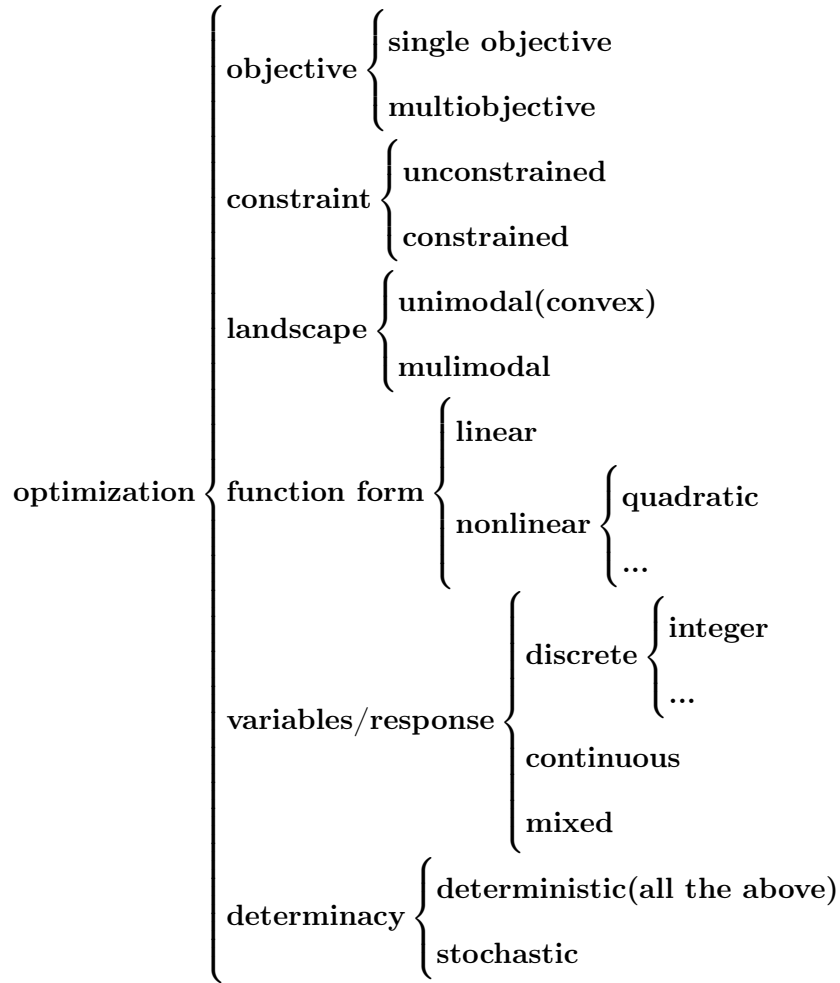


Figure 2.1: Classification of optimization problems

the values in both design variables and objective/constraints are exact, and there is no uncertainty or noise in their values. All the optimization problems are deterministic in the sense that for any given set of design variables, the values of both objective functions and the constraint functions are determined exactly. Thus, if there are any uncertainty and noise in the design variables and the objective functions and/or constraints, then the optimization becomes a stochastic optimization problem, or a robust optimization problem with noise.

2.3 Computational complexity

The difficulty of an optimization problem is generally measured mathematically in the form of complexity theory of which the central question is what makes some problems computationally hard and others easy. Thus, complexity theory is part of the theory of computation dealing with minimal resources required during computation to solve a given problem. The amount of computational resources is determined by its time and space complexity [Rothlauf 2011, Edelkamp 2012].

- *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.

The main objective of complexity theory is to deal with the following tasks [Avazbeigi 2009]:

- Define clearly what solving a problem efficiently means.
- Categorize problems into those that can be solved efficiently and those that cannot.
- Estimate the amount of time (or memory) needed to solve problems.

It is clear that the efficiency of an algorithm refers to its usage of computer resources and the usual measure is the algorithm's running time which clearly depends on the size and nature of the input. However, there are some resolution techniques whose behavior can vary even for a fixed input as in randomized algorithms or among inputs of the same size. A randomized algorithm is one that makes some random (or pseudorandom) choices.

In the next section, we recall some basic concepts from complexity theory which are used throughout the thesis and may be useful to understand the complexity of problem solving methods.

2.3.1 Definitions and basic concepts

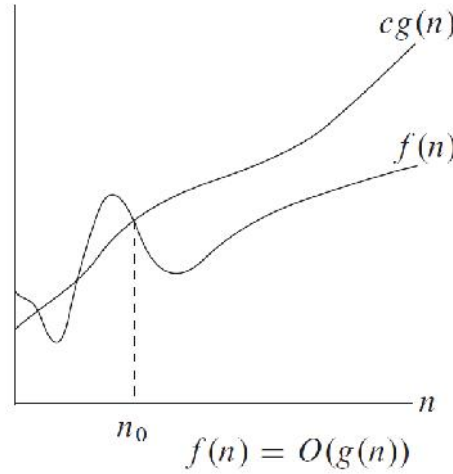
Definition 1.1 [Korte 2012, Levitin 2012] An algorithm is a sequence of unambiguous instructions for solving a problem, i.e., for obtaining a required output for any legitimate input in a finite amount of time. Each instruction can be composed of elementary steps (basic operations), such that for each valid input the computation of the algorithm is a uniquely defined finite series of elementary steps which produces a certain output.

However, one is not usually satisfied with a finite computation but rather want a good upper bound on the number of elementary steps performed, depending on the input size.

Definition 1.2. (The Big-O Notation) [Cormen 2009] Let $f, g : D \rightarrow \mathbb{R}^+$ be two functions of variable n . We say that f is $O(g)$ if $f(n)$ is bounded above by some constant multiple of $g(n)$ for all large n , that is, if there exist positive constants c and n_0 such that $0 \leq f(n) \leq cg(n)$ for all $n \geq n_0$.

Note that for all values n at and to the right of n_0 , the value of the function $f(n)$ is on or below $cg(n)$ as illustrated in Figure 2.2. Thus, the Big-O notation is used to describe concisely the running time of an algorithm by finding only the worst-case running time, that is, the longest running time for any input of size n , because the exact running time of an algorithm is often a complex expression and it is usually estimated.

Let us take an example, consider a function defined as $f(n) = 2n^2 + 5n + 3$. It is easy to show that $f(n) = O(n^2)$. We have $f(n) = 2n^2 + 5n + 3$, so that, $f(n) \leq 2n^2 + 5n^2 + 3n^2 = 10n^2$ for all $n \geq 1$. Hence, there exists $c = 10$ and $n_0 = 1$ such that $f(n) \leq cn^2$ for all $n_0 \geq 1$.

Figure 2.2: Graphic example of the O notation.

Definition 1.3. (The time complexity of an algorithm) Let A be an algorithm which accepts inputs from a set X , and let $f : \mathbb{N} \rightarrow \mathbb{R}^+$. If there exist constants $c, n_0 > 0$ such that A terminates its computation after at most (worst-case running time) $cf(n)$, for all $n \geq n_0$, elementary steps for each input $x \in X$ and n denotes the size of x , then we can say that A runs in $O(f)$ time or the running time (or the **time complexity**) of A is $O(f)$.

Definition 1.4. (Polynomial time) An algorithm is said to run in polynomial time if there is a fixed positive integer k such that its time complexity is $O(n^k)$, where n is the input size. Note that running times like $O(n \log n)$ are also polynomial time, since $n \log n = O(n^2)$. Algorithms running in linear ($O(n)$), logarithmic ($O(\log n)$) and constant time ($O(1)$) are the faster algorithms of those running in polynomial time. A computational problem is said to be solved efficiently if it is solvable in polynomial time. The problems that can be solved in polynomial time are also scalled *tractable problems* or *easy problems*. Otherwise, the problems that cannot be solved in polynomial time are called *intractable problems*.

Definition 1.5. (Exponential time) An algorithm is said to run in exponential time if there is a fixed positive integer k such that its time complexity

is $O(k^n)$, where n is the input size. As the size of the problem increases linearly, the time to solve the problem increases exponentially.

Definition 1.6. (An instance of a problem) [Cormen 2009] An instance I of a problem P consists generally of the input (satisfying whatever constraints that are imposed in the problem statement) needed to compute a solution to the problem. This means that I is obtained by specifying particular values for all parameters of P .

Definition 1.7. (Performance guarantee) [Korte 2012] Let P be an optimization problem with non-negative weights and $k \geq 1$. A k -factor approximation algorithm for P is a polynomial-time algorithm A for P such that

$$\frac{1}{k} \text{OPT}(I) \leq A(I) \leq k \text{OPT}(I) \quad (2.4)$$

for all instances I of P . We also say that A has **performance ratio** (or **performance guarantee**) k . The first inequality applies to maximization problems while the second one to minimization problems. Note that for instances I with $\text{OPT}(I) = 0$ we require an exact solution. The 1-factor approximation algorithms are precisely the exact polynomial-time algorithms.

2.3.2 Complexity classes

The subject of computational complexity theory deals with the classification of problems according to their level of difficulty. As stated previously, the computational complexity of a problem is usually measured in terms of the minimal computational resources required for its solution. This depends generally on the time of the best found algorithm that can solve this problem. Therefore, a complexity class is a set of problems of related complexity. As depicted in Figure 2.3, four important classes of problems are usually identified in this context: \mathcal{P} , \mathcal{NP} , \mathcal{NP} -hard, and \mathcal{NP} -complete classes. For a general background on the complexity classes presented here, see [Karp 1972, Avazbeigi 2009, Garey 1979]. Recall that a decision problem is a problem that

has only two possible answers: *yes* or *no*. For example, the problem of deciding whether a graph has a Hamilton cycle. A Hamilton cycle of a graph is a simple cycle that passes through each vertex of the graph.

2.3.2.1 Class \mathcal{P}

It is the set of all decision problems that can be solved by polynomial-time algorithms. Note that \mathcal{P} stands for polynomial time. For all problems in \mathcal{P} , an algorithm exists that can solve any instance of the problem in time that is $O(n^k)$.

2.3.2.2 Class \mathcal{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 \mathcal{NP} stands for non-deterministic polynomial time. An equivalent definition of \mathcal{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, \mathcal{P} is a subset of \mathcal{NP} , $\mathcal{P} \subseteq \mathcal{NP}$. One of the important open questions in complexity theory is whether class \mathcal{P} is equal or not equal class \mathcal{NP} . Nevertheless, this has never been proven and it is widely believed that $\mathcal{P} \neq \mathcal{NP}$.

2.3.2.3 Class \mathcal{NP} -complete

Assume that $\mathcal{P} \neq \mathcal{NP}$, a subset of $\mathcal{NP} \setminus \mathcal{P}$ comprising the hardest problems in \mathcal{NP} are called *\mathcal{NP} -complete problems*. Formally, A problem is *\mathcal{NP} -complete* if it is in \mathcal{NP} and each \mathcal{NP} problem can be reduced to it in polynomial time. This class has a significant property that if any \mathcal{NP} -complete problem can be solved in polynomial time, then every problem in \mathcal{NP} has a polynomial-time solution, that is, $\mathcal{P} = \mathcal{NP}$. Despite years of study, no polynomial-time

algorithm has ever been found for any \mathcal{NP} -complete problem [Cormen 2009].

2.3.2.4 Class \mathcal{NP} -hard

The class of \mathcal{NP} -hard problems contains problems which are not in \mathcal{NP} but any \mathcal{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 \mathcal{NP} . It is a more general class of problems that comprises, for example, the search and optimization variants of the \mathcal{NP} -complete problems. However, The fact that an optimization problem is \mathcal{NP} -hard does not imply that all instances are difficult to solve. Whereas in practice there are at least three ways of dealing with these problems [Hoos 2004]:

- Find an application relevant subclass of the problem that can be solved efficiently. Optimization problems which are \mathcal{NP} -hard may still be possible to have important special cases that are solvable in polynomial time.
- If we are faced with a case where a polynomial solution does not exist, one possibility is to accept suboptimal solutions in polynomial time with a certain performance guarantee using efficient approximation algorithms. Formally, the degree of suboptimality of a solution quality is expressed in the form of the approximation ratio as described in the previous section.
- Another possibility when the notion of approximation cannot be applied at all, is to focus on stochastic rather than deterministic algorithms.

We should note that these three possible ways summarize the types of optimization methods that can be used to cope with this class of problems. They are explained briefly in the following sections.

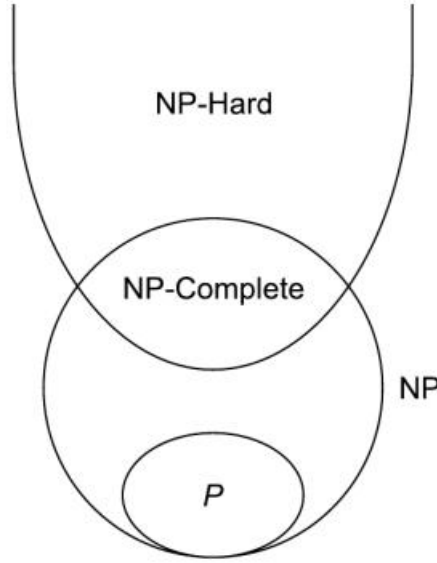


Figure 2.3: The relationships among complexity classes of problem (Assuming $P \neq NP$).

2.4 Local Versus Global Optima

Solutions found by optimization algorithms are classified by the quality of the solution. The main types of solutions are referred to as *local optima* or *global optima*. In the following discussion, it will be assumed, without loss of generality, that by optimization we mean minimization. Let us consider the problem of optimizing (minimizing) a function (objective function) $f(x)$ subject to $x \in \mathcal{F} \subseteq \mathcal{S}$ where \mathcal{F} is the set of feasible solutions. The point (solution) $x^* \in \mathcal{F}$ is a *global optimum* (*global minimum*) of f if $f(x^*) \leq f(x)$ for all $x \in \mathcal{F}$. Therefore, the global minimum is the best of a set of candidate solutions. We should note that an optimization problem may have more than one global optimum. In addition to the global minimum, the function f may also admit local minimum (minima). A point $x_{\mathcal{N}}^* \in \mathcal{F}$ is a local minimum of f if there exists an ε -neighbourhood set $\mathcal{N} \subseteq \mathcal{S}$, around $x_{\mathcal{N}}^*$ such that $f(x_{\mathcal{N}}^*) \leq f(x)$ for each $x \in \mathcal{F} \cap \mathcal{N}$, for some $\varepsilon > 0$ (i.e., $\mathcal{N} = \{x : \|x_{\mathcal{N}}^* - x\| \leq$

$\varepsilon\}$). It is obviously that $\mathcal{F} \cap \mathcal{N}$ represents the set of feasible points in the neighborhood of $x_{\mathcal{N}}^*$. Figure 2.4 [Bazaraa 2006] illustrates instances of local and global optimum.

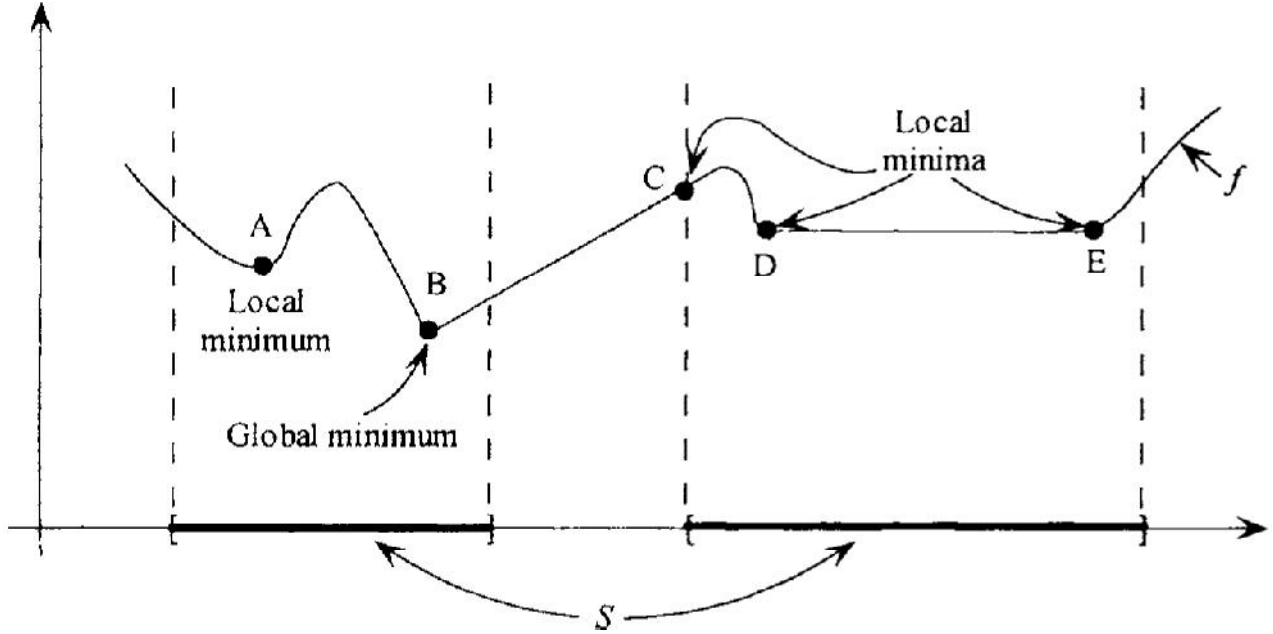


Figure 2.4: Local Versus Global Optima

2.5 Optimization methods

In this thesis we concentrate on combinatorial optimization problems which is by far the most popular type of optimization that has attracted the interest of many researchers in the recent years. Combinatorial optimization problems are conceptually easy to model but most of them are quite difficult to solve in practice. The main difficulty in solving combinatorial optimization problems arises from the fact that the number of feasible solutions from which the best solutions needs to be selected grows exponentially with the size of the problem instances. Let us consider the classical combinatorial optimiza-

tion problem known as the traveling salesman problem that involves finding the shortest path tour visiting each of N cities exactly once and returning to the starting city. An instance of the problem with N cities contains $\frac{(N-1)!}{2}$ solutions. Hence, enumerating all of these solutions is of complexity of the order of $O(N^N)$. One possibility to deal with this problem is to perform an exhaustive search which enumerates the entire set of feasible solutions and evaluates the cost of each solution, after which the best solution is retained. For example, if $N = 10$ there are 1.81×10^5 possible solutions. Enumerating all of them and choosing the best one is feasible in a short computation time. However, if $N = 100$ there are about 4.66×10^{155} solutions. In this case, enumerating all the possible solutions seem to require a large computation time. Consequently, exhaustive search is only practical for problems with a relatively small number of cities.

Combinatorial optimization problems can be solved by using two different types of approaches that are briefly presented in the remaining part of this section (see Fig. 2.5 [El-Ghazali 2009]). The first approach is by using exact algorithms. The second approach is applied to more complex or larger problems, leads to near-optimal solutions and is characterized by the use of approximate algorithms. Approximate algorithms can be further divided to heuristic and meta-heuristic methods.

Usually, if for an approximate algorithm it can be proved that it returns solutions that are worse than the optimal solution by at most some fixed value or factor, such an algorithm is also called an *approximation algorithm* [Dorigo 2004].

2.5.1 Exact methods

Exact methods (also named as complete algorithms) are guaranteed to find an optimal solution in bounded time for every finite size instance of a combina-

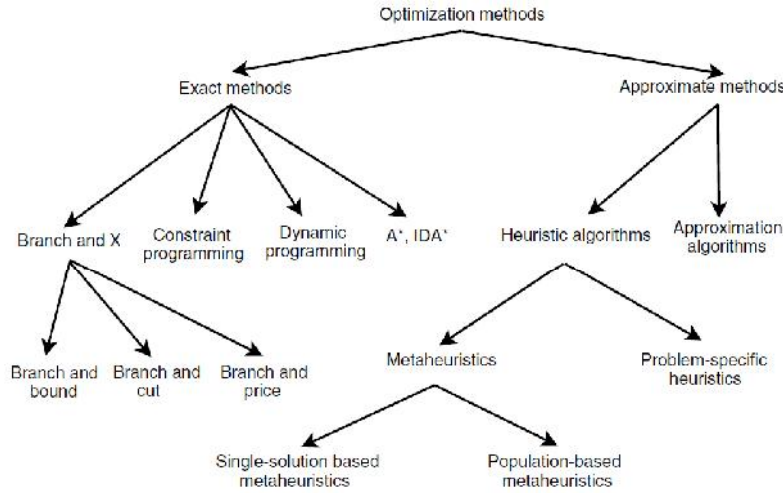


Figure 2.5: Classical optimization methods

torial optimization problem and assesses its optimality. However, for the typical combinatorial optimization problems which are usually difficult to optimize, no algorithms exist to solve these problems in polynomial time. Furthermore, the computation time for these methods in most cases increases exponentially with respect to the problem size, leading to impractical computation time for real large-scale problems. Therefore, exact methods are usually suitable for computations on small instances. Examples of some well-known exact methods include the family of Branch and X (Branch and Bound algorithm, Branch and Cut algorithm and Branch and Price algorithm), Dynamic Programming, Linear Programming and as well as many specialized algorithms that have been developed for particular combinatorial optimization problems.

2.5.2 Approximate methods

An unfortunate reality is that, real-life combinatorial optimization problems are typically of big size and the number of solutions can grow exponentially with the size of the problem, so scanning all solutions one by one and selecting the best one is not practical due to the computational expense of per-

forming an exhaustive search. In other words, they cannot be solved exactly within a reasonable amount of computational resources. Therefore, using approximate approaches is the main alternative to solve this class of problems [El-Ghazali 2009]. According to Talbi [El-Ghazali 2009], approximate algorithms can be classified in two families: specific heuristics and metaheuristics.

2.5.2.1 Heuristics

The word heuristic has its origin in the old Greek verb *heuriskein*, which means *to find*. A heuristic is a technique which seeks good (i.e. near-optimal) solutions at a reasonable computational cost without being able to guarantee either feasibility or optimality, or even in many cases to state how close to optimality a particular feasible solution is [Reeves 1993].

Heuristic is also defined by Yang [Yang 2010] as a solution strategy by trial-and-error to produce acceptable solutions to a complex problem in a reasonably practical time. The complexity of the problem of interest makes it impossible to search every possible solution or combination, the aim is to find good, feasible solutions in an acceptable timescale. There is no guarantee that the best solutions can be found, and we even do not know whether an algorithm will work and why if it does work.

The idea is that an efficient but practical algorithm that will work most of the time and be able to produce good quality solutions. Among the found quality solutions, it is expected that some of them are nearly optimal, though there is no guarantee for such optimality.

On the other hand, Heuristics are often problem-specific, so that a method which works for one problem cannot be used to solve a different one. The usual classification of heuristic methods is either constructive or local search methods [Blum 2003, Dorigo 2004] .

- Constructive algorithms generate solutions from scratch by iteratively adding solution components to an initially empty solution until the solution is complete. Constructive heuristics are generally the fastest approximate algorithms although some special implementations may induce a high computational load. Their advantage in computational time requirements is counterbalanced by generally inferior quality solutions when compared to local search techniques.
- Local search algorithms start from an initial solution (most of the times generated by a constructive heuristic or randomly) and iteratively try to replace part or the whole current solution with a better one in an appropriately defined set of neighboring solutions by local changes. The definition of a neighborhood is important for optimization problems as it determines which solutions are similar to each other. A neighborhood is a mapping $\mathcal{N}(x) : X \rightarrow 2^X$, where X is the search space containing all possible solutions to the problem. 2^X stands for the set of all possible subsets of X and \mathcal{N} is a mapping that assigns to each element $x \in X$ a set of elements $y \in X$. That is, a neighborhood definition can be viewed as a mapping that assigns to each solution $x \in X$ a set of solutions y that are neighbors of x [Rothlauf 2011].

2.5.2.2 Metaheuristics

A disadvantage of single-run algorithms like constructive methods or iterative improvement is that they either generate only a very limited number of different solutions, which is the case for greedy construction heuristics, or they stop at poor quality local optima, which is the case for iterative improvement methods. Unfortunately, the obvious extension of local search, that is, to restart the algorithm several times from new starting solutions, does not produce significant improvements in practice. Several general approaches, which

are nowadays often called *metaheuristics*, have been proposed which try to bypass these problems [Dorigo 2004]. The term *metaheuristic* was first introduced by Glover [Glover 1986]. It is composed of two Greek words, *Heuristic* and the suffix *meta* which means *beyond, in an upper level*. Before this term was widely adopted, metaheuristics were often called *modern heuristics*.

Although there is no commonly accepted definition of the term metaheuristic, the following definition seems to be most appropriate: a metaheuristic is an iterative master process that guides and modifies the operations of subordinate heuristics to efficiently produce high quality solutions by combining intelligently different concepts for exploring and exploiting the search space. The subordinate heuristics may be high (or low) level procedures, or a simple local search, or just a construction method [Osman 1996, Glover 1997].

Unlike exact optimization algorithms, metaheuristics do not guarantee the optimality of the obtained solutions. Instead of approximation algorithms, metaheuristics do not define how close are the obtained solutions from the optimal ones.

Based on the above definition, the main properties characterizing the notion of metaheuristic can be summarized in the following points [Blum 2003].

- 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.
- Metaheuristic algorithms are approximate and usually non-deterministic.
- They may incorporate mechanisms to avoid getting trapped in confined areas of the search space.

- The basic concepts of metaheuristics permit an abstract level description.
- Metaheuristics are not problem-specific.
- Metaheuristics may make use of domain-specific knowledge in the form of heuristics that are controlled by the upper level strategy.
- Today's more advanced metaheuristics use search experience (embodied in some form of memory) to guide the search.

Metaheuristics have received increasing attention in recent years in the solution of combinatorial optimization problems, since they are able to yield high quality solutions with reasonable computational effort. The family of metaheuristics includes, but is not limited to, Ant Colony Optimization (ACO), Genetic Algorithms (GA), Iterated Local Search (ILS), Simulated Annealing (SA), Tabu Search (TS), Particle Swarm Optimization (PSO), Bee Colony algorithm (BCA), Artificial Immune Systems (AIS), Bacterial Foraging Algorithm (BFA), Greedy Randomized Adaptive Search Procedure (GRASP) and their hybrids.

2.6 Classification of metaheuristics

In this section we briefly summarize the most common ways of classifying metaheuristics [El-Ghazali 2009]

- **Nature inspired versus non-nature inspired:** perhaps the most intuitive way to classify metaheuristics may be based on the origin of the algorithm which can be classified as nature-inspired versus or not nature-inspired. GA, AIS, and BFA are examples of metaheuristics inspired from biology. ACO, BCA, and PSO are inspired from the collaborative

behavior of social animals (swarm intelligence). SA is inspired from the physical annealing process.

- **Deterministic versus stochastic** A deterministic metaheuristic solves an optimization problem by making deterministic decisions (for example TS). In stochastic metaheuristics, some random rules are applied during the search (for example GA, ACO, and PSO). In deterministic algorithms, using the same initial solution will lead to the same final solution, whereas in stochastic metaheuristics, different final solutions may be obtained from the same initial solution. This characteristic must be taken into account in the performance evaluation of metaheuristic algorithms.
- **Population-based search versus single-solution based search** Another possibility of classification is based on the number of candidate solutions maintained by the metaheuristic simultaneously. Single-solution based algorithms (for example SA and ILS) manipulate and transform a single solution during the search while in population-based algorithms (for example GA, ACO, and PSO) a whole population of solutions is evolved. These two families have complementary characteristics: single-solution based metaheuristics are exploitation oriented. They have the power to intensify the search in local regions. Population-based metaheuristics are exploration oriented since they allow a better diversification in the whole search space.
- **Iterative versus greedy** In iterative algorithms, we start with a complete solution (or population of solutions) and transform it at each iteration using some search operators. Greedy algorithms start from an empty solution, and at each step a decision variable of the problem is assigned until a complete solution is obtained. Most of the metaheuristics

are iterative algorithms.

Stochastic optimization techniques

Contents

3.1	Search techniques	32
3.1.1	Blind search strategies	32
3.1.2	Heuristic search strategies	32
3.2	Iterated local search	33
3.3	Greedy Randomized Adaptive Search Procedure . .	36
3.4	Genetic algorithm	38
3.5	Ant colony Optimization	40
3.5.1	Swarm intelligence	40
3.5.2	The Basic Principle of Ant Colony Optimization . . .	41
3.5.3	Solution construction	43
3.5.4	local improvement	44
3.5.5	Pheromone update	44
3.6	Motif finding using ant colony optimization	45
3.6.1	The Motif Finding Problem	47
3.6.2	Basic Motif Representations	48
3.6.3	The Basic Gibbs Sampling Algorithm	51
3.6.4	The Proposed Approach (MFACO)	53
3.6.5	Computational Experiments	57
3.6.6	Concluding remarks	60

The aim of this chapter is to provide a quick description of the main stochastic optimization techniques that are related to this work, such as iterated local search, GRASP, genetic algorithm and ant colony optimization. At the end of the chapter, we explain how ant colony optimization can be applied successfully to provide the motif finding problem with promising solutions.

3.1 Search techniques

Search techniques have been widely used to solve many optimization problems of both theoretical and practical importance. It is one of the most universal problem solving methods for such problems in which one cannot determine a priori the sequence of steps leading to a solution [Gen 2000]. There are important issues in the discussion of search strategies: exploring the search space and exploiting the best solution [Booker 1987]. Therefore, search can be performed with either blind strategies or heuristic strategies.

3.1.1 Blind search strategies

Blind strategies are used in cases where little or no information is available about the problem to be solved. Random search is an example of a strategy which explores the search space while ignoring the exploitation of the promising regions of the search space. It has also a better ability to escape from a local optimum.

3.1.2 Heuristic search strategies

Heuristic strategies are used when certain information about the problem is provided. Hill-climbing is an example of a heuristic strategy which exploits

the best solution for possible improvement at the expense of ignoring the exploration of the search space. It uses additional information to guide search move along with the best search directions. Furthermore, it is capable of climbing upward toward a local search optimum. In the design of such meta-heuristic approaches, attention should be paid to two contradictory criteria: exploration of the search space (diversification) and exploitation of the best solutions found (intensification)[[El-Ghazali 2009](#)]. It is obvious that combining elements of direct and random search can lead to a significant balance between exploration and exploitation of the search space.

3.2 Iterated local search

Iterated local search (ILS) [[Lourenço 2003](#)] belongs to the class of stochastic local search algorithms [[Hoos 2004](#)] that extend classical local search methods by adding diversification capabilities to allow them to escape from local optima on their way towards the global optimum.

The basic idea of ILS is to perform a biased, randomized walk in the space of locally optimal solutions instead of sampling the space of all possible candidate solutions. This walk is built by iteratively applying first a perturbation to a locally optimal solution, then applying a local search algorithm, and finally using an acceptance criterion which determines to which locally optimal solution the next perturbation is applied.

A crucial component of ILS is the strength of the perturbation which determines the amount of changes made on the current solution. If it is too strong, ILS may behave like a random restart resulting in a very low probability of finding better solutions. On the other hand, if the perturbation is not strong enough, the local search procedure will rapidly go back to a previous local optimum. Despite the simplicity of ILS, it is on the basis of several state-of-the-art algorithms for real-world problems [[Hoos 2004](#)].

The principle of the ILS is pictorially illustrated in Figure 3.1. To design an ILS algorithm, four components have to be defined [Paquete 2002]:

- A procedure that generates an initial solution s_0 .
- A perturbation mechanism that modifies the current solution (local optimum) s leading to some intermediate solution s^p which is the starting solution of the next local search in order to escape from local optima and to explore new regions of the search space. There are different strategies that can be used to handle the perturbation strength during the search [El-Ghazali 2009]:
 - *Static perturbation*: the perturbation length is fixed a priori (independent of the instance size) before the search and is no longer modified during the search. For more difficult problems, it has been shown that fixed-strength perturbations may lead to poor performance.
 - *Dynamic perturbation*: the perturbation length is determined dynamically during the search without taking into account the search history.
 - *Adaptive perturbation*: the perturbation length is determined dynamically during the search according to some information about the search history.
- A local search procedure that returns an improved solution s' . A basic local search algorithm works in an iterative fashion by successively replacing the current solution by a better solution in the neighborhood of the current solution. It terminates when no better solution is found in the neighborhood. The pseudo-code of a basic local search algorithm given in Algorithm 1.

- An acceptance criterion procedure that decides to which solution the next perturbation is applied.

A general pseudocode for ILS algorithm is outlined in Algorithm 3.1.

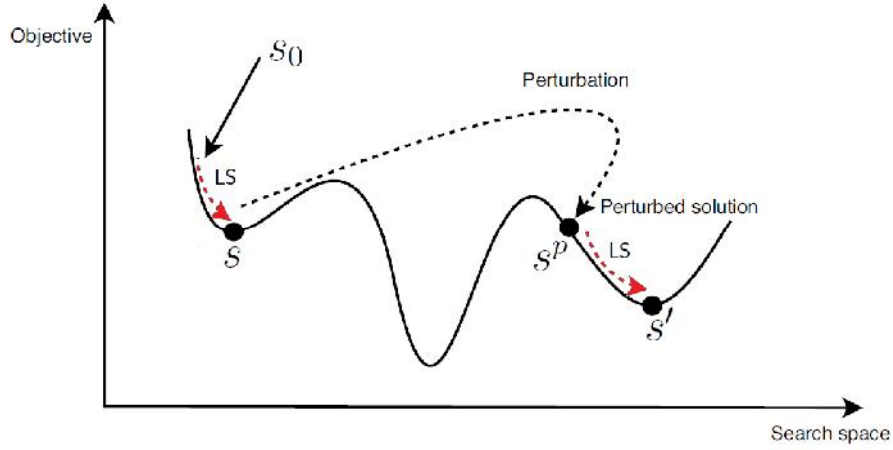


Figure 3.1: The principle of ILS. Starting with a local minimum s , a perturbation is performed, leading to a solution s^p . After applying a local search procedure (LS), we may find a new local minimum s' that is better than s .

Algorithm 1 Local Search

- 1: **Input:** s
 - 2: **while** s is not locally optimal **do**
 - 3: Find $s_{\mathcal{N}}^b \in \mathcal{N}(s)$ such that $s_{\mathcal{N}}^b < f(s)$ // Suppose that we have a minimization problem and f is the objective function.
 - 4: $s \leftarrow s_{\mathcal{N}}^b$
 - 5: **output:** s
-

Algorithm 2 Iterated Local Search

```

1:  $s_0 \leftarrow \text{Generate\_Initial\_Solution}()$  // initial solution
2:  $s \leftarrow \text{Local\_Search}(s_0)$ 
3: while termination conditions not met do
4:    $s^p \leftarrow \text{Perturbation}(s, \text{history})$ 
5:    $s' \leftarrow \text{Local\_Search}(s^p)$ 
6:    $s \leftarrow \text{Acceptance\_Criterion}(s, s', \text{history})$ 
7: output:  $\{s$  // best solution found in the search process  $\}$ 

```

3.3 Greedy Randomized Adaptive Search Procedure

GRASP [Feo 1995] (Greedy Randomized Adaptive Search Procedure) is an iterative approach that combines the power of greedy heuristics, randomization and local search procedures. In general, each iteration consists of two phases: a (greedy adaptive randomized) construction phase and a local search improvement phase.

The construction phase is itself an iterative greedy and adaptive process since the greedy function takes into account previous decisions in the construction when considering the next choice. Besides, there is a probabilistic component in GRASP that is applied to the selection of elements during the construction phase. A solution is usually represented as a set of elements. The construction phase builds an initial solution by considering one element at a time. It starts from an empty set and iteratively adds elements to it until a feasible complete solution is generated. Each element is added using a greedy function that optimizes the objective function. At each stage of the construction, elements that have not yet been selected are ordered and a list called *restricted candidate list* (RCL), recording the highest quality elements with respect to an adaptive greedy function, is formed. Then, the probabilistic component is

determined by randomly choosing from RCL the next element to be added which not necessarily corresponds to the best one in the list. We should note that repeated applications of the construction procedure yields different constructed feasible solutions.

The solutions generated by the construction phase are not guaranteed to be locally optimal considering neighborhood concepts. The local search phase attempts to improve each constructed solution by means of a local search algorithm that iteratively replaces the current solution with a better one.

The GRASP procedure stops when a specified termination condition is reached, for example, after completing a certain number of iterations and the best solution found overall the different iterations is returned as output. Algorithm 3 depicts the pseudo-code for a GRASP procedure. As the performance of most stochastic optimization algorithms depends mainly on the balance between diversification and intensification, The α threshold is used to control the amount of greediness of the construction mechanism. In [Wang 2010], a value-based scheme¹ is applied to form a RCL based on the value of α . For better understanding, let us consider an objective function $f(x) : X \rightarrow \mathbb{R}$ to maximize, and X its solution space. So, we look for a solution $x^* \in X$ with $f(x^*) \geq f(x)$ for all $x \in X$. In this case, a RCL can be defined as

$$RCL = \{x \in X | f(x) \geq f(x)_{min} + \alpha(f(x)_{max} - f(x)_{min})\} \quad (3.1)$$

where $\alpha \in [0, 1]$, $f(x)_{max} = \max\{f(x) | x \in X\}$, $f(x)_{min} = \min\{f(x) | x \in X\}$. Obviously, $\alpha = 0$ corresponds to a pure greedy construction while $\alpha = 1$ is equivalent to a random construction.

¹ There is another mechanism that can be used to generate a RCL (named cardinality-based RCL) where the k best rated solution elements are selected to be in the list

Algorithm 3 GRASP

```

1: Input:  $\alpha$ 
2:  $s \leftarrow \text{Construct\_Random\_Solution}()$ 
3: while termination conditions not met do
4:    $s^p \leftarrow \text{Greedy\_Randomized\_Construction}(\alpha)$ 
5:    $s \leftarrow \text{LocalSearch}(s^p)$ 
6:    $s \leftarrow \text{Acceptance\_Criterion}(s^p, s)$ 
7: output:  $\{s \text{ // best solution found in the search process } \}$ 

```

3.4 Genetic algorithm

Genetic algorithm (GA) [Goldberg 1989, Gen 2000] is a class of general purpose stochastic search method inspired by the mechanisms of natural selection and natural genetics representing them in three principal genetic operators : *selection*, *crossover* and *mutation*. GAs combine exploitation (selection) of the best solutions from past searches with the exploration (crossover and mutation) of new regions of the solution space. Each potential candidate solution in the search space of the problem is represented by an individual or (chromosome) whose quality is measured according to a fitness function.

A general structure of a GA is depicted in Algorithm 4. It begins with creating the initial population where each individual can be generated either randomly or using some heuristic methods. Then, GA maintains a population of individuals $P(t)$ that evolves over generations. At each generation t of GA, a new population $P(t + 1)$ is generated from the old population $P(t)$ through selection and reproduction (crossover and mutation).

Selection is the process of selecting individuals for reproduction on the basis of their relative fitness (quality measure). An individual with high fitness value has a higher probability of survival and being selected, and so the greater the amount of its genetic material that will be passed on to future generations. There are several methods for performing selection (For more

details see [Goldberg 1989]). *Roulette wheel selection* is the most common and easy-to-implement selection mechanism. It is a proportional selection strategy in which slots of a virtual roulette wheel are sized according to the fitness of each individual in the population. Spinning the roulette wheel and noting the position of the pointer select an individual. The probability of selecting an individual is therefore proportional to its fitness. It is clear that individuals with the largest fitness (and slot sizes) have more of a chance to be chosen.

Roulette wheel selection has several disadvantages [Freitas 2002]. Firstly, it assumes that all individuals have either positive or zero fitness values and that the fitness function must be maximized, rather than minimized. Otherwise, it has to be modified to respect these assumptions. Secondly, it necessitates the computation of a global statistic (the total sum of fitness values of all individuals in the population), which requires an additional computational effort. Thirdly, if one individual has a fitness value much better than the others, it will tend to be selected much more often than the others. If this individual is a local optimum there will be a *premature convergence* for that suboptimal solution rather than a global optimum. This drawback, called premature convergence in the terminology of GAs, occurs if the population of a GA reaches such a suboptimal state that the genetic operators are no longer able to produce offspring that are able to outperform their parents. This happens if the genetic information stored in the individuals of a population does not contain that genetic information which would be necessary to further improve the solution quality [Affenzeller 2005]. Fourthly, if most individuals have about the same fitness value (which often occurs in later generations, when the EA is converging), those individuals will have about the same probability of being selected, so that the selection will be almost random.

For the reproduction, some of the selected individuals undergo stochastic transformations by means of the two genetic operators, crossover and muta-

tion, to form a new population of individuals called *offspring* $C(t)$, which are then evaluated. The crossover operator creates new individuals by combining parts (exchanging information) from two individuals called parent individuals. The mutation operator creates new individuals by making changes in a single individual. It is used to introduce diversity in the population, which is very useful to spread the search to an unexplored region of the search space. A new population $P(t+1)$ is formed by selecting the more fit individuals from the parent population $P(t)$ and the offspring population $C(t)$. After several generations, the algorithm converges to the best individual, which hopefully represents an optimal or suboptimal solution to the problem [Gen 2000].

Algorithm 4 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 // Reproduc-
      tion
6:   evaluate  $C(t)$ 
7:   select  $P(t+1)$  from  $P(t)$  and  $C(t)$  // Selection
8:    $t \leftarrow t + 1$ 

```

3.5 Ant colony Optimization

3.5.1 Swarm intelligence

Swarm intelligence is a modern artificial intelligence discipline that is concerned with the design of intelligent multi-agent systems by taking inspiration from the collective behavior of social insects such as ants, termites, bees, and wasps, as well as other animal societies such as flocks of birds or schools of

fish. They are expected to exhibit the features that may have made social insects so successful in the biosphere [Bonabeau 2001]:

- *Flexibility*, which means that the group can quickly adapt to a changing environment.
- *Robustness*, so that even when one or more individuals fail, the group can still perform its tasks.
- *Self-organization*, which means that the group needs relatively little supervision or top-down control.

Optimization and swarm robotics are nowadays two of the domains where swarm intelligence principles have been applied very successfully. A third and very popular application domain concerns routing and load-balancing in telecommunication networks [Blum 2008].

3.5.2 The Basic Principle of Ant Colony Optimization

Ant Colony Optimization (ACO)[Dorigo 1992, Dorigo 1996, Dorigo 2004] is one of the first swarm-intelligence techniques, which was introduced by Marco Dorigo and colleagues in the early 90s, proposed for providing approximate solutions to hard combinatorial optimization problems. ACO is inspired by the foraging behavior of real ant colonies, specifically the indirect communication, via pheromone trails, between the ants regarding a shortest path between their nest and a food source in the environment. When searching for food, ants initially explore the area surrounding their nest in a random manner. While moving, ants deposit a certain amount of chemical substance called pheromone on the ground, thus forming a pheromone trail. Ants can smell the pheromone and they tend to choose, probabilistically, paths marked by strong pheromone concentrations. As soon as an ant finds a food source, it evaluates the quantity and the quality of the food and carries some of it back to the nest. During the

return trip, the quantity of pheromone that an ant leaves on the ground may depend on the quantity and quality of the food. The pheromone trails will guide other ants to the food source [Dorigo 2004, Blum 2008]. That is, the higher the pheromone concentration found on a particular path, the more the number of ants that go through the path, then the more the pheromone left by ants. Furthermore, the pheromones have the property of evaporation over time, so if a path is not being followed by ants, the pheromones evaporate and the path loses its attraction to ants and disappears over time. In analogy to the biological example, ACO is on indirect communication within a colony of simple agents, called (artificial) ants, mediated by (artificial) pheromone trails which serve as a distributed, numerical memory that the ants use to probabilistically construct candidates solutions to the problem being solved. The pheromone trails are adapted during the search process and represent the collective search experience of the ants [Dorigo 2003].

A general framework of the ACO metaheuristic is outlined in Algorithm 5. After initializing parameters and pheromone trails, the main loop consists of three main phases: at each iteration, a number of solutions are constructed by the ants. These solutions are then improved through a local search (this step is optional), and finally the pheromone is updated [Dorigo 2006]. In the context of ACO, solutions and partial solutions are assembled as sequences of solution components taken from a finite set of solution components $C = \{c_1, \dots, c_{n_c}\}$ (where $n_c = |C|$).

Algorithm 5 Ant Colony Optimization

- 1: Set parameters, initialize pheromone trails
 - 2: **while** termination conditions not met **do**
 - 3: Construct_Ants_Solution
 - 4: Update_Pheromone
 - 5: Apply_Local_Serach // Optional
-

3.5.3 Solution construction

Each ant of the colony concurrent constructs solutions to the problem instance under consideration. An ant is implemented as a stochastic construction procedure that probabilistically build a candidate solution s in an incremental way starting with an empty partial solution $s^p = \emptyset$ and iteratively adding a feasible solution component c_i , from the set $N(s^p) \subset C$, without backtracking to the current partial solution s^p until a complete solution is obtained. At each construction step, the choice of a solution component is performed using a stochastic rule, which is biased by two factors:

- *Heuristic information* (greedy heuristic) on the problem instance being solved, if available, which can guide the ants towards the most promising solutions. That is, a desirability value τ_{c_i} (heuristic information) is assigned at each construction step to each solution component c_i .
- *Pheromone trails* which change dynamically at run-time to reflect the ants acquired search experience. That is, a pheromone value η_{c_i} is associated with each solution component c_i . It is initially set to some constant value at the beginning of the ACO algorithm.

A stochastic component in ACO allows the ants to build a wide variety of different solutions and hence explore a much larger number of solutions than greedy heuristics. In most ACO algorithms, The rule for the stochastic choice of the next solution component, also called the transition probabilities, is defined as follows:

$$P(c_i|s^p) = \frac{[\tau_{c_i}]^\alpha [\eta_{c_i}]^\beta}{\sum_{c_j \in N(s^p)} [\tau_{c_j}]^\alpha [\eta_{c_j}]^\beta}, \forall c_i \in N(s^p) \quad (3.2)$$

where $N(s^p)$ is the set of feasible components. The value of parameters α and β , $\alpha > 0$ and $\beta > 0$, control the relative importance of pheromone versus the heuristic information.

3.5.4 local improvement

Once the solutions have been constructed, and before updating the pheromone, it is common to improve the solutions obtained by the ants through a local search. This phase, which is highly problem-specific, is optional although it is usually included in state-of-the-art ACO algorithms.

3.5.5 Pheromone update

The aim of the pheromone update is to increase the pheromone values associated with good or promising solutions, and to decrease those that are associated with bad ones. It consists of two parts:

- Decrease uniformly all the pheromone values through pheromone evaporation. Pheromone evaporation is the process by means of which the pheromone deposited by previous ants decreases over time. From a practical point of view, pheromone evaporation is needed to avoid a too rapid convergence of the algorithm towards a suboptimal region. It implements a useful form of forgetting, favoring the exploration of new areas of the search space.
- Increase the pheromone values associated with solution components that are part of one or more good solutions from either the current or earlier iterations.

A large number of different ACO algorithms have been proposed. They generally differ from each other in the pheromone update rule that is applied. The first one proposed in the literature, called Ant System [Dorigo 1996], in which the pheromone update rule that was used is defined as

$$\tau_{c_j} = (1 - \rho) \cdot \tau_{c_j} + \rho \cdot \sum_{\{s \in \mathcal{S}_{iter} | c_i \in s\}} F(s) \quad (3.3)$$

The parameter $\rho \in (0, 1]$ is called *evaporation rate*. \mathcal{S}_{iter} is the set of solutions constructed in the current iteration. We assume that we are dealing with a minimization problem. $F : \mathcal{S} \mapsto \mathbb{R}^+$ is a function, commonly called the *quality function*, such that $f(s) < f(s') \Rightarrow F(s) \geq F(s'), \forall s \neq s' \in \mathcal{S}$.

Another successful ACO variant is $\mathcal{MAX}\text{-}\mathcal{MIN}$ Ant System (MMAS). MMAS [Stützle 2000] is an improved ant system, which is characterized by the following three main points:

- Pheromone trails are updated after each iteration, only the best ant is allowed to deposit its pheromones. The best ant can be chosen as either the best ant of the current iteration or the global best ant (the best-so-far ant). The results described in the literature bias the choice towards the first one.
- Pheromone trails values are restricted to a lower and an upper bound $[\tau_{min}, \tau_{max}]$ in order to avoid stagnation behavior suffered by other ant systems. This stagnation behavior is a consequence of reinforcing some solution components by laying more pheromone on them unlike the others that are rarely used. This behavior leads to a repeated selection of the same solution while other regions of the search space are not explored.
- Pheromone trails are initialized to the maximum trail value τ_{max} to encourage exploring a large area of the search space at the beginning of the algorithm.

3.6 Motif finding using ant colony optimization

Finding the location of the common motif, shared by a set of DNA sequences, in each sequence has become a fundamental problem in Bioinformatics with

important applications in locating regulatory sites and drug target identification [Pevzner 2000]. The motif finding problem has been formally considered as a difficult pattern recognition problem. Most developed motif finding algorithms use either approximate or heuristic techniques to obtain near optimal solutions at relatively low computational cost. Some of them carry out the search in the space of possible starting positions, whereas others search in the space of all possible motifs based on a given model. Recent researches covering most of the relevant techniques and approaches for motif finding, as well as several of the benchmark algorithms included in this work can be found in [Das 2007, Tompa 2005]. Moreover, bio-inspired algorithms and other meta-heuristics have been also proposed. Examples of these algorithms include genetic algorithms [Liu 2004, Che 2005, Kaya 2009], genetic programming [Seehuus 2005], and simulated annealing [Keith 2002]. Although these methods have been shown to generate acceptable results in terms of the quality of the solutions found, the motif finding problem is still unsolved. In [Liao 2003], an ant colony optimization algorithm was developed to find a set of better initial positions for the Gibbs sampler (GS) [Lawrence 1993] in order to improve its efficiency in term of time computing and score. However, it does not incorporate any form of heuristic information. Moreover, a specific ant colony system was used for predicting the MHC class II binders [Karpenko 2005]. In this study, we present a new motif finding approach based on ant colony optimization (ACO) called MFACO that combines $\mathcal{MAX-MIN}$ Ant System [Stützle 2000], one of the best performing variants of ACO metaheuristic, with a modified form of the original GS playing the role of a local heuristic optimization step since most ACO algorithms developed in literature incorporate a particular local optimizer to improve the produced solutions. Unlike some other motif finding techniques, MFACO searches both in the space of starting positions as well as in the space of motif patterns. Due to this feature, it has more chances to find potential motif patterns. Although our approach is

also valid for protein sequences, we apply it only to DNA sequences. The rest of the section is organized as follows. First, we formally introduce the motif finding problem. Second, our proposed approach for tackling the problem is presented. Finally, we discuss the conducted experiments and we give some concluding remarks.

3.6.1 The Motif Finding Problem

A formal description of this problem can be viewed as follows. Given a set of DNA sequences $\mathcal{S} = \{S_1, S_2, \dots, S_N\}$ of common length W ². Find the promising motif pattern $X = x_1x_2\dots x_l$ of length l , $x_i \in \{A, T, C, G\}$ and the starting locations of its occurrences on all sequences in \mathcal{S} . The selection of a particular motif pattern is based on a defined score function that measure the similarity between the motif pattern and its occurrences. There are several methods for scoring a motif pattern. Our proposed approach uses *consensus score* [Jones 2004] and *information content* [Che 2005, Stormo 1989] as score functions. To illustrate how to compute these score functions, consider a candidate motif pattern that can be generated by choosing a random position from each sequence. Then, the patterns starting at these positions are aligned to form an $N \times l$ alignment matrix. Therefore, the candidate motif pattern can be represented by a count-based profile C where $C(r, j)$ is the count of nucleotide r on the column j of the alignment matrix and its corresponding consensus score (CS_c) is defined as:

$$CS_c = \sum_{j=1}^l \left(\max_{r \in \{A, T, C, G\}} (C(r, j)) \right) \quad (3.4)$$

²Just for the purpose of simplicity the set of sequences tested in our algorithm are with the same length.

The information content (IC) score function can be easily computed as follows:

$$IC = \sum_{j=1}^l \sum_{r \in \{A,T,C,G\}} Q(r, j) \cdot \log_2 \frac{Q(r, j)}{B_0(r)} \quad (3.5)$$

Where each element $Q(r, j)$ indicates the frequency of the nucleotide r to be in position j of the motif pattern and $B_0(r)$ denotes its background frequency, i.e. the observed frequency of nucleotide r overall all sequences in the dataset.

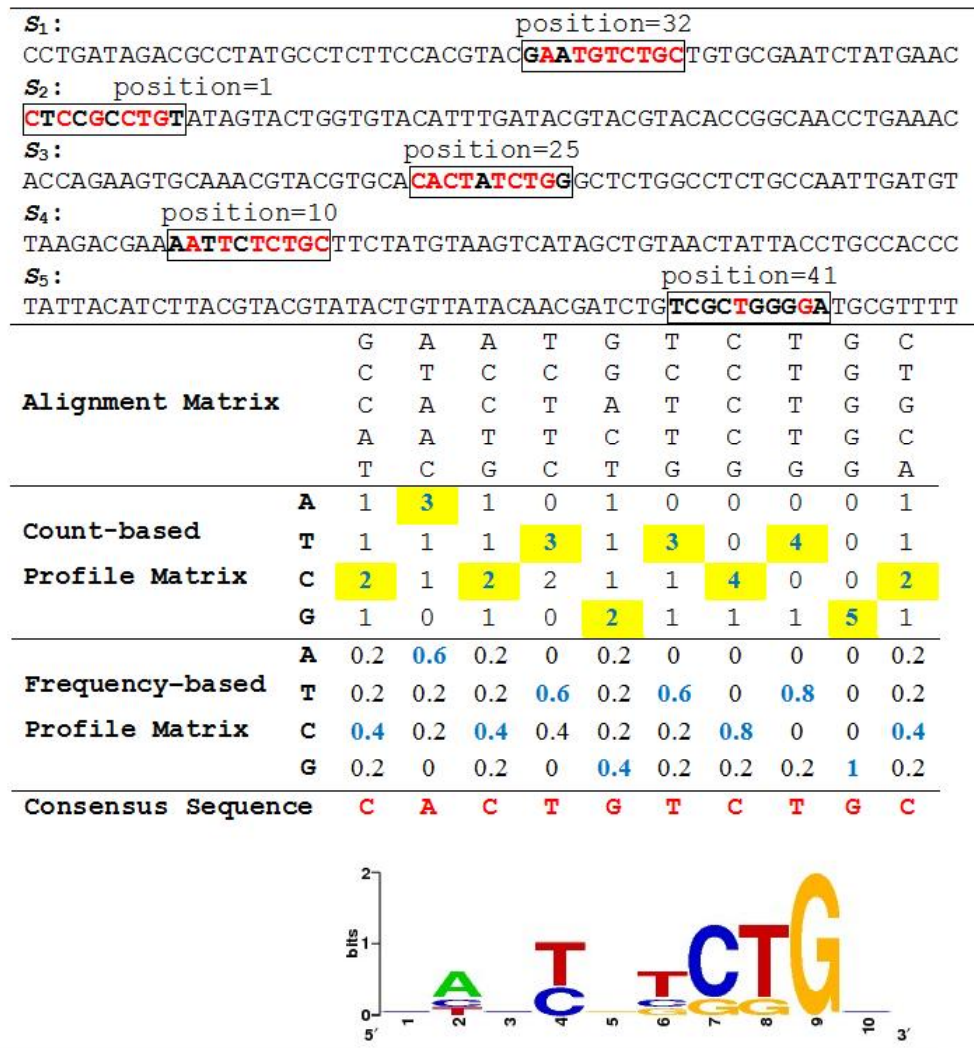
3.6.2 Basic Motif Representations

Motif finding algorithms perform their search either in *the space of possible starting positions* of all motif occurrences in the sample sequences or in *the space of motif patterns* described by a given model. Example of the latter approach includes the pattern driven approach (i.e., exhaustively enumerating all possible patterns) and the sample driven approach (i.e., selecting some patterns from the sample sequences to be used as seeds for local search) [Price 2003]. Besides, two standard models are generally used to represent a motif of length either by $4 \times l$ profile matrix or by a pattern of l consensus sequence.

3.6.2.1 Profile Model

The common profile model to describe a motif of length l uses a position probability matrix, $4 \times l$ profile matrix Q , as given in Eq. 3.6, where each entry $Q(r, j)$ gives the probability that an occurrence of the motif contains nucleotide r in its j^{th} position.

$$Q = \begin{bmatrix} q_{A,1} & q_{A,2} & \cdots & q_{A,l} \\ q_{T,1} & q_{T,2} & \cdots & q_{T,l} \\ q_{C,1} & q_{C,2} & \cdots & q_{C,l} \\ q_{G,1} & q_{G,2} & \cdots & q_{G,l} \end{bmatrix} \quad (3.6)$$



Thus, the profile matrix, or simply profile, illustrates the variability of nucleotide composition at each position from a particular choice of candidate motif [Jones 2004]. A profile, in its simplest form, can be viewed as the frequency of nucleotides in each position of a particular alignment. Figure 3.6.2.1 gives an example that illustrates how to represent a particular motif using a frequency-based profile. Consider a sample of sequences $\mathcal{S} = \{S_1, S_2, S_3, S_4, S_5\}$ with an unknown implanted motif of length ($l=10$). A candidate motif can be generated by selecting one random position in each of these sequences. Let GAATGTCTGC, CTCCGCCTGT, CACTATCTGG, AATTCTCTGC and TCGCTGGGGA be the patterns starting at these positions respectively. To compute its associated profile matrix Q , firstly, patterns are aligned to form an alignment matrix. Then, a $4 \times l$ count-based profile C is constructed. Each element $C(r, j)$ is defined as the number of times (letter counting) nucleotide $r \in \{A, T, C, G\}$, appears in column j of the alignment matrix. Finally, the frequency-based profile Q is the result of dividing each element of C by the number of sequences ($N=5$) in the current sample.

3.6.2.2 consensus Model

In this model, consensus sequence summarizes the profile matrix by a pattern of l consensus sequence representing the most frequent nucleotide (the consensus nucleotide) in each motif position, that is, the nucleotide with the largest entry in the profile matrix (see Figure 3.6.2.1 for a given example). However, a profile is a more flexible representation than consensus sequence because the latter says nothing about how strongly the consensus nucleotide in each motif position is conserved or about the distribution of any non-consensus nucleotides.

3.6.3 The Basic Gibbs Sampling Algorithm

Gibbs sampling is a Markov Chain Monte Carlo algorithm, randomized algorithm that has seen wide application in the statistical community. It was first applied in Bioinformatics as a tool for finding motif in 1993 [Lawrence 1993]. Since that, Gibbs sampling has become a well-known method for finding motifs in DNA sequences. Gibbs sampling operates as follows:

Input: Given a set of N DNA sequences $\mathcal{S} = \{S_1, S_2, \dots, S_N\}$ and an unknown implanted motif of fixed length l . Assume that there is exactly one motif instance located within each sequence.

Output: A set $\mathfrak{s} = \{s_1, s_2, s_3, \dots, s_N\}$ represents the starting positions of the chosen patterns in each of the N sequences such that the similarity between them is maximized (the similarity is measured using a particular score function explained later in this section). Thus, the obtained motif is simply the consensus sequence of these patterns.

Data structure: In addition to the set of starting positions, the algorithm maintains two data structures:

- *A pattern description* in the form of a profile model Q , i.e., a $4 \times l$ profile matrix ($4 \times l$ profile matrix in the case of protein sequences) represented in term of nucleotide frequencies.
- *A probabilistic description of the background frequencies* $B = [b_A b_T b_C b_G]$ ($|B| = 20$ when Gibbs sampling is applied to protein sequences). The background model is computed from the input sequences. Each element b_r , $r \in \{A, T, G, C\}$, denotes the frequency of nucleotide r in dataset excluding the sites described by the motif pattern. By the way, most improvements of the Gibbs sampling algorithm are due to the use of advanced background models (see [Thijs 2001, Liu 2001] for more details).

Basic steps:

- (1) Choose a random starting position s_k within each input sequence S_k , where $S_k \in \mathcal{S}$, $1 \leq k \leq N$ and $1 \leq s_k \leq n_k - l + 1$ (n_k is the length of S_k). Let $\mathcal{X} = \{X_1, \dots, X_k, \dots, X_N\}$ where X_k represents the pattern of length l starting at position s_k within sequence S_k .
- (2) Randomly select one sequence S_{out} , whose pattern is X_{out} and its associated starting position is s_{out} , to be excluded (out) from \mathcal{S} . Let $\mathcal{S}^* = \mathcal{S} - \{S_{out}\}$ and $\mathcal{X}^* = \mathcal{X} - \{X_{out}\}$.
- (3) Create a profile Q from \mathcal{X}^* . The profile matrix is computed as follows

$$Q(r, j) = \frac{C(r, j) + u_r}{N - 1 + U} \quad (3.7)$$

where $C(r, j)$ is the count of nucleotide r in position j after the alignment of motif patterns in \mathcal{X}^* . u_r is a pseudocount whose value is proportional to the frequency of nucleotide r in all the dataset, while their sum U is computed by

$$U = \sum_{r \in \{A, T, C, G\}} u_r \simeq \sqrt{N} \quad (3.8)$$

Analogously, each element b_r of the background frequencies B is given is given by Eq. 3.9.

$$b_r = \frac{c_r + u_r}{\sum_r c_r + U} \quad (3.9)$$

where c_r represents the count of nucleotide r taken over all non pattern positions from \mathcal{S}^* .

- (4) For each position s_{temp} in S_{out} , $1 \leq s_{temp} \leq n_{out} - l + 1$ (n_{out} is length of S_{out}), calculate the probability P_{temp} (using Eq. 3.10) that the pattern X_{temp} of length l starting at position s_{temp} is generated by profile Q and background probabilities B .

$$P_{temp} = \frac{P(X_{temp}/Q)}{P(X_{temp}/B)} \quad (3.10)$$

If $X_{temp} = x_1 x_2 \cdots x_l$ then $P(X_{temp}/Q)$ and $P(X_{temp}/B)$ are calculated by Eq. 3.11 and Eq. 3.12 respectively.

$$P(X_{temp}|Q) = \prod_{j=1}^l Q(x_j, j) \quad (3.11)$$

$$P(X_{temp}|B) = \prod_{j=1}^l B(x_j) \quad (3.12)$$

- (5) Choose the new starting position s_{temp} in S_{out} randomly, with probability proportional to P_{temp} . Therefore, s_{out} is replaced by s_{temp} .
- (6) From the updated set of starting positions $\mathfrak{s} = \{s_1, s_2, s_3, \dots, s_N\}$, the score function F , that Gibbs sampling algorithm maximizes, is calculated as follows.

$$F = \sum_{j=1}^l \sum_{r \in \{A, T, C, G\}} C(r, j) \cdot \log_2 \frac{Q(r, j)}{b_r} \quad (3.13)$$

where $C(r, j)$ and $Q(r, j)$ are calculated as in Step (3) except that they are calculated from the complete alignment, i.e., including the excluded sequence.

- (7) If stopping condition is verified (The algorithm may stop either after a certain number of iterations or when it reaches a state that keeps generating the same near optimal discovered motif pattern) then display the current set of starting positions $\mathfrak{s} = \{s_1, s_2, s_3, \dots, s_N\}$ and their corresponding motif pattern and stop the algorithm, else go to Step (1).

3.6.4 The Proposed Approach (MFACO)

Our developed approach, called Motif Finding based on Ant Colony Optimization (MFACO), has the same general framework of an ACO algorithm as shown in Algorithm 5. For more discussion of the basic steps involved in the design of an ACO see Section 3.5. In next subsections, we detail each of the MFACO components.

3.6.4.1 Initialization

The initialization step of any ACO algorithm includes three fundamental tasks:

- Choosing an appropriate graph representation for the problem to be solved.
- Defining the problem specific heuristic information.
- Representing the pheromone trail model including the manner to set the initial pheromone trails.

Choosing the appropriate graph representation for the problem to be solved is of central importance to graph-based ACO metaheuristic. In our case, we use a weighted directed graph $G(V, E)$ with V being the set of nodes and E being the set of edges. If the motif's length is l then there are $4 \times l$ nodes arranged in a grid of four rows and l columns. Each node in position (i, j) , simply denoted by $node(i, j)$, is associated with nucleotide i to be in the j^{th} position of the motif. In addition, an edge $e_j(u, v)$ always exists between two nodes $node(u, j)$ and $node(v, j + 1)$ where $u, v \in \{A, T, C, G\}$ and $(1 \leq j \leq l - 1)$ as shown in Fig. 3.3. In MFACO, two types of pheromone trails are modeled. First, a pheromone trail τ_i^1 , $i \in \{A, T, C, G\}$, is associated with each $node(i, 1)$. The value τ_i^1 encodes the desirability of nucleotide i being in the first motif's position. Second, a pheromone trail τ_{uv}^j , $u, v \in \{A, T, C, G\}$, is associated with each edge $e_j(u, v)$. The value τ_{uv}^j corresponds to the desirability of nucleotides u and v being at motif's position j and $j + 1$ respectively. Initially, the pheromone values are all set to a constant value.

3.6.4.2 Solution Construction

MFACO consists of a number of iterations of solution construction. Each ant incrementally builds its solution by traversing the graph to complete a

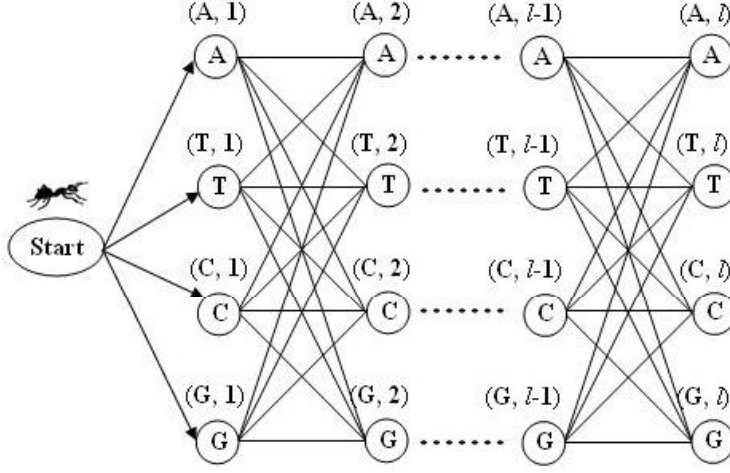


Figure 3.3: Problem graph-based representation

tour representing one candidate motif pattern. The probability $P_i^1(t)$ of an ant selects $node(i, 1)$ as the first solution component, at iteration t , can be expressed as:

$$P_i^1(t) = \frac{[\tau_i^1(t)]^\alpha [\eta_i^0]^\beta}{\sum_{u \in \{A, T, C, G\}} [\tau_u^1(t)]^\alpha [\eta_u^0]^\beta} \quad (3.14)$$

where η_i^0 is the heuristic information of $node(i, 1)$ representing the frequency of nucleotide i in the dataset, that is, $\eta_i^0 = B_0(i)$. Besides, an ant located at $node(u, j)$ chooses to go to $node(v, j + 1)$ with a probability defined as:

$$P_{uv}^j(t) = \frac{[\tau_{uv}^j(t)]^\alpha [\eta_{v,j}^n]^\beta}{\sum_{r \in \{A, T, G, C\}} [\tau_{ur}^j(t)]^\alpha [\eta_{r,j}^n]^\beta} \quad (3.15)$$

where $\eta_{v,j}^n$ is the heuristic information of edge $e_j(u, v)$ which is computed based on the higher-order background model introduced in [Thijs 2001]. Using a background model of order n means that, in addition to pheromone information, the ant's decision to add a solution component (nucleotide) i in position j of the pattern motif depends on the n previous visited nucleotides during the solution construction. Doing so, the n^{th} background model B_n is based on counting the frequency of all nucleotide subsequences of length $(n + 1)$ in the dataset. Let $x_1 x_2 \dots x_{j-1} x_j$ be the partial motif pattern con-

structed by an ant and being at $node(u, v)$, i.e. $x_j = u$. If $j \geq n$ then $\eta_{v,j}^n = P(x_{j+1} = v | x_{j-n+1} \dots x_{j-1} x_j)$, otherwise $\eta_{v,j}^n = B_0(v)$. After each ant builds its solution and before the pheromone update, the generated solution is improved by applying the GS technique that plays the role of a local heuristic optimization step. It differs from the original model in the following two aspects: (i) the sequence to be excluded from the DNA sample is chosen in a round robin manner but not randomly; (ii) the new starting position with the highest score is selected instead to be randomly chosen with probability proportional to its calculated score. By these two modifications, the stochastic behavior of GS is greatly reduced and it is to the ACO that the task is shifted and preserved.

3.6.4.3 Pheromone Update

The values of pheromone trail τ_i^1 , at each iteration t , are updated according to:

$$\tau_i^1(t+1) = (1 - \rho) \cdot \tau_i^1(t) + \Delta\tau_i^{1,best} \quad (3.16)$$

where ρ ($0 < \rho < 1$) is the pheromone evaporation rate and $\Delta\tau_i^{1,best}$ is the amount of pheromone trail deposited on $node(i, 1)$ given by

$$\Delta\tau_i^{1,best} = Q^{best}(i, 1) \cdot CSc^{best} / (N \cdot l) \quad (3.17)$$

where N is the number of sequences in the dataset and CSc^{best} is the consensus score of the best solution found either in the current iteration or so far by the algorithm which is the case of MFACO. The pheromone trail τ_{uv}^j on each edge $e_j(u, v)$ is updated according to the following updating rule:

$$\tau_{uv}^j(t+1) = (1 - \rho) \cdot \tau_{uv}^j(t) + \Delta\tau_{uv}^{j,best} \quad (3.18)$$

where

$$\Delta\tau_{uv}^{j,best} = Q^{best}(u, j) \cdot Q^{best}(v, j+1) \cdot CSc^{best} / (N \cdot l) \quad (3.19)$$

Similar to $\mathcal{MAX-MIN}$ Ant System [Stützle 2000], MFACO also limits the pheromone values into an interval $[\tau_{\min}, \tau_{\max}]$ with $\tau_{\max} = CSc^{best}/((1-\rho) \cdot N \cdot l)$ and $\tau_{\min} = 0$.

3.6.5 Computational Experiments

MFACO was implemented in Matlab[®] version 6.5. It stops either when it reaches a predetermined maximum number of iterations or when stagnation state [Dorigo 1996] occurs, that is, all ants keep generating the same motif pattern. Our experimental results were obtained on a PC with Pentium[®]IV (2.4 GHz) and 256 MB of memory. The performance of the proposed algorithm is tested on the following biological samples. Firstly, a collection of three datasets, that have been used as benchmark data in [Liu 2004] with 6, 9, 18 sequences respectively. Each sequence has an equal length of 3001 nucleotides. Secondly, we have also used the sample of experimentally confirmed E. coli CRP binding sites [Stormo 1989]. This dataset consists of 18 sequences. Each sequence has a length of 105 nucleotides and contains at least one CRP-binding site. The conserved motif was located in 23 binding sites using the DNA Footprinting method (FP), with a motif length of 22.

The results generated by MFACO were compared with those obtained using some recent algorithms previously described for finding potential motifs. The algorithms include MotifSampler (MS) [Thijs 2001], BioProspector (BP) [Liu 2001], MEME [LcBailey 1994], and Genetic Algorithms (GAs). We are mainly interested in GA-based approaches, such as FMGA [Liu 2004] and MDGA [Che 2005], since our approach is also a population based approach that uses stochastic choices to guide its search. FMGA searches in the space of motif patterns and uses consensus score as a similarity measure, while MDGA was implemented using information content score function and searches in the space of starting positions.

First collection of datasets. For comparison reasons we use the same way of presenting the results as in [Liu 2004] and the effectiveness of such approach is measured with respect to the consensus score. The computational results for MEME, MS, and BP are obtained by sending the three datasets to each approach's website to return the three best-found motifs. The motif length is set to $l = 7$ and $l = 13$. After some preliminary experiments, we found that it is suitable to initialize MFACO using the following parameter settings: $it_{\max} = 50$, $n = 6$, $m = 8$, $\alpha = 2$, $\beta = 1$, $\rho = 0.15$ and all pheromone trail values are initially set to one. Table 3.1 through Table 3.6 list the best motif patterns discovered by the investigated approaches in each dataset. In each table, the first column gives the names of the approaches including ours. The second and third columns indicate the consensus sequence and the consensus score(CSc) of the found motif respectively. The column $s = 0$, $s = 1$, and $s = 2$ denotes the total number of sequences in the dataset that completely match the corresponding motif (i.e., exact matching without any mismatch), at most one mismatch, and at most two mismatches respectively. Table 3.1 shows that all approaches were able to report one completely matched motif ($CSc = 42$) of length $l = 7$. MS, MEME, and FMGA return only two completely matched motifs from the three top ranked patterns. The three best motif patterns found by BP represent the same completed matched motifs. Otherwise, MFACO achieves better results than the other approaches since it can find 18 completely matched motifs from the beginning. The use of a hash matrix by MFACO facilitates the task of finding motifs that match all sequences especially when n is set to 6. From the rest of the tables we can see also that the motif patterns with the highest accuracy are generally found by MFACO while the others fail to discover them. We should also mention that MFACO can find different potential motif patterns having the same consensus score as well as those with multiple occurrences in a single sequence. Besides,

it is clear that the serial implementation of our approach takes more time but not at the cost of an exhaustive search which has to scan the set of all 4^l patterns for a motif of length l . During our experiments, at most three runs for MFACO seems to be sufficient to predict better motifs in most cases within an acceptable computational time. The average running time was about 15 seconds, 60 seconds, 30 seconds, 75 seconds, 95 seconds, and 200 seconds for the following tests (Dataset 1, $l = 7$), (Dataset 1, $l = 13$), (Dataset 2, $l = 7$), (Dataset 2, $l = 13$), (Dataset 3, $l = 7$), and (Dataset 3, $l = 13$) respectively.

E. Coli CRP Binding Sites. This dataset was used to test the performance of MFACO compared to MDGA [Che 2005]. In doing so, we adjusted MFACO to work with the information content score rather than the consensus score, which is calculated according to Eq. (3.5). Each zero element of the frequency-based profile $Q(r, j)$ is set to $(0.25/23)$ and $B_0(r) = 0.25$ for each nucleotide r following the same setting done in [Stormo 1989]. Table 3.7 summarizes the starting positions of motifs discovered in the CRP dataset using FP, GS, BP, and MDGA respectively. A single sequence may contain two occurrences of the same motif. Additional column ER follows each method shows the deviation of the predicted starting positions from the exact starting positions. From the presented results shown in Table 3.7, it is clear that all the three approaches (GS, BP, and MDGA) failed to predict the exact starting positions identified by Footprinting. However, Table 3.8 illustrates that MFACO is capable of finding the exact binding sites (i.e., BS3) and new binding sites (BS) BS1, BS2, and BS4 representing the same referenced motif. Table 3.9 lists the consensus sequence of motifs discovered by each approach and its corresponding relative entropy score. In [Stormo 1989], it was stated that the most highly conserved nucleotides in the CRP binding sites constitute a consensus sequence of `nnnTGTGAnnnnnnnTCACA`. Positions for which no consensus nucleotide was found (with frequency less than 50) are reported as

n. The obtained results in Table 3.8 show that the four sets of binding sites found by MFACO maintain this constraint. Consequently, MFACO achieves the higher prediction accuracy

3.6.6 Concluding remarks

As conclusions, this study was to investigate and adapt ACO for identifying potential motifs in gene promoter regions. The proposed approach uses a modified version of the Gibbs sampler playing the role of local optimizer. Experimental results have shown that MFACO is able to generate better potential motif patterns in term of prediction accuracy than other methods such as GAs, MEME, MS, and BP within a reasonable computation time. Moreover, the proposed algorithm differs from other motif finding techniques by searching both in the space of starting positions and in the space of motif patterns. This combination significantly provides more chance to explore the search space. MFACO commonly maintains its diversity until a near optimal solution will be found and can be easily adapted to work with different score functions such as consensus score and information content.

During our experiments, based on the CRP dataset, a central issue in finding the correct binding sites (BS) is selecting the appropriate parameters of the relative entropy score function that reveals the relationship between the highest score and the correct BS. In other words, a candidate BS with highest relative entropy calculated according to specific settings may not necessarily correspond to the correct BS (for example the settings used by [Che 2005]). In fact, the score function should be carefully chosen before implementing any motif finding algorithm.

Table 3.1: Dataset 1, $l = 7$

<i>Method</i>	<i>Consensus</i>	<i>CSc</i>	<i>s = 1</i>	<i>s = 0</i>
MFACO	AAAAAAA	42	6/6	6/6
	AGGAGGA	42	6/6	6/6
	AAAAAAG	42	6/6	6/6
	TAAAAAT	42	6/6	6/6
MS	GCGGGCG	42	6/6	6/6
	CGCCGCC	42	6/6	6/6
	GGGGCGG	41	6/6	5/6
BP	GCCGCCG	42	6/6	6/6
MEME	AAAAAAA	42	6/6	6/6
	TAAAAAT	42	6/6	6/6
	AAATAAA	41	6/6	5/6
FMGA	AAAAAAA	42	6/6	6/6
	AGGAGGA	42	6/6	6/6

Table 3.2: Dataset 1, $l = 13$

<i>Method</i>	<i>Consensus</i>	<i>CSc</i>	<i>s=2</i>	<i>s=1</i>
MFACO	AAAAAAAAAAAAAGA	76	6/6	6/6
	AAAAAAAAAAAAAAG	75	6/6	5/6
	AAAAAAAAAAAAAGT	75	6/6	6/6
	AAAAAGAAAAAGA	75	6/6	5/6
MS	CGCCGCCGCCGCC	72	6/6	4/6
	CCGCCGCCGCCGC	71	6/6	3/6
	CGGCGGGCGGGGG	70	6/6	4/6
BP	GCCGCCGCCGCCG	72	6/6	4/6
MEME	AAAAAAAAAAAAAGA	76	6/6	6/6
	GAGGCTGAGGCAG	71	5/6	4/6
	AAAAAAAAAAAAAGA	76	6/6	6/6
FMGA	AAAAAAAAAAAAA	73	6/6	4/6

Table 3.3: Dataset 2, $l = 7$

<i>Method</i>	<i>Consensus</i>	<i>CSc</i>	$s = 1$	$s = 0$
MFACO	CCCTCCT	63	9/9	9/9
	CCCTCAG	62	9/9	8/9
	GGGTTGG	62	9/9	8/9
	GAGCAGG	62	9/9	8/9
MS	CCCGGGC	61	9/9	7/9
	CCGCGCC	60	9/9	6/9
	CGGGCGC	59	9/9	5/9
BP	GAGACGG	59	9/9	5/9
	AGTAACT	58	9/9	4/9
MEME	AAAAAAA	60	9/9	6/9
	AGGAAGA	59	9/9	5/9
	TTTTTTT	58	9/9	4/9
FMGA	CCCTCCT	63	9/9	9/9
	GGGCTGG	62	9/9	8/9

Table 3.4: Dataset 2, $l = 13$

<i>Method</i>	<i>Consensus</i>	<i>CSc</i>	$s=2$	$s=1$
MFACO	GCCGGCGGGCGCC	102	8/9	4/9
	GCGGGCGGGCGCC	101	9/9	2/9
	GCCGGCGGGCGGC	100	7/9	3/9
	GCCGGAGGGCGCC	100	8/9	2/9
MS	GGCCCCCGGGCGG	101	7/9	3/9
	CCCCGCCCCGGC	100	8/9	2/9
	GGGGGCGCCGGG	99	7/9	4/9
BP	GGCCCCCGGGCGG	101	7/9	3/9
	GGCCCGCGGGCGG	100	6/9	4/9
	CGGCCCCGCGCGG	97	5/9	2/9
MEME	GAAAGAGAAAGG	99	6/9	4/9
	GGGGAGGTGGAGT	96	5/9	1/9
	TTTATTTTATTTT	95	5/9	2/9
FMGA	GCGGGGCGCGGG	101	6/9	4/9
	GGCCGGGCGCGGG	100	6/9	5/9

Table 3.5: Dataset 3, $l = 7$

<i>Method</i>	<i>Consensus</i>	<i>CSc</i>	$s = 1$	$s = 0$
MFACO	GGGGCGG	123	18/18	15/18
	CCCAGCT	123	18/18	15/18
	CCAGCTG	123	18/18	15/18
	CTGAGGC	123	18/18	15/18
MS	GGCGGGG	123	18/18	15/18
	GCGGGGC	121	18/18	13/18
	GCGCGGG	119	18/18	11/18
BP	CGCAGGC	115	18/18	7/18
	CTGTGAT	115	18/18	7/18
	CTTGAAC	114	18/18	6/18
MEME	GTCTCTA	116	17/18	9/18
	GGCTCAA	114	18/18	6/18
	TTATCAG	105	13/18	2/18
FMGA	GGTGAGG	122	18/18	14/18
	AAAAAAA	119	18/18	11/18

Table 3.6: Dataset 3, $l = 13$

<i>Method</i>	<i>Consensus</i>	<i>CSc</i>	$s=2$	$s=1$
MFACO	GGGAGGCTGAGGC	205	16/18	6/18
	CGGGAGGCGGAGG	204	15/18	7/18
	GGAGGCTGAGGCA	202	12/18	7/18
	CGGGAGGCGGGGG	201	15/18	7/18
MS	CGGGGGGCGGAGG	196	12/18	4/18
	GGGCCGGGCGCGG	195	13/18	3/18
	GGGCCGGGCGGGG	195	11/18	4/18
BP	GAAACTCCGTCTC	186	6/18	5/18
	GCGAAACCCGTC	186	7/18	4/18
	GCGAAACTCCGTC	185	6/18	5/18
MEME	AAAAAAAAAAAAA	194	11/18	4/18
FMGA	GGGAGGCGGAGGC	201	13/18	9/18
	AAAAAACAAAAAA	196	11/18	7/18








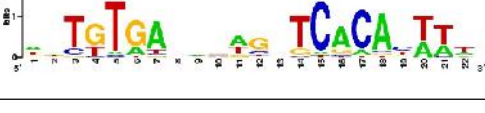
Table 3.7: Performance results achieved by GS, BP, and MDGA in the CRP dataset [Che 2005].

<i>Seq.</i>	<i>FP</i>	<i>GS</i>	<i>ER</i>	<i>BP</i>	<i>ER</i>	<i>MDGA</i>	<i>ER</i>
1	17, 61	59	-2	63	2	62	1
2	17, 55	53	-2	57	2	56	1
3	76	74	-2	78	2	77	1
4	63	59	-4	65	2	64	1
5	50	11	-39	52	2	51	1
6	7, 60	5	-2	9	2	8	1
7	42	40	-2	26	-16	43	1
8	39	37	-2	41	2	40	1
9	9, 80	7	-2	11	2	10	1
10	14	12	-2	16	2	15	1
11	61	59	-2	63	2	62	1
12	41	47	6	43	2	42	1
13	48	46	-2	50	2	49	1
14	71	69	-2	73	2	72	1
15	17	15	-2	19	2	18	1
16	53	49	-4	55	2	54	1
17	1, 84	25	24	68	-16	56	-28
18	78	74	-4	80	2	77	1

Table 3.8: Performance results achieved by MFACO in the CRP dataset.

<i>Seq.</i>	<i>FP</i>	<i>BS1</i>	<i>ER</i>	<i>BS2</i>	<i>ER</i>	<i>BS3</i>	<i>ER</i>	<i>BS4</i>	<i>ER</i>
1	17, 61	61	0	61	0	61	0	61	0
2	17, 55	55	0	55	0	55	0	55	0
3	76	76	0	76	0	76	0	76	0
4	63	63	0	63	0	63	0	63	0
5	50	50	0	50	0	50	0	50	0
6	7, 60	7	0	7	0	7	0	7	0
7	42	24	-18	42	0	42	0	42	0
8	39	39	0	39	0	39	0	20	-19
9	9, 80	9	0	9	0	9	0	9	0
10	14	14	0	14	0	14	0	14	0
11	61	61	0	61	0	61	0	61	0
12	41	41	0	41	0	41	0	41	0
13	48	48	0	48	0	48	0	48	0
14	71	71	0	71	0	71	0	71	0
15	17	17	0	17	0	17	0	17	0
16	53	53	0	53	0	53	0	53	0
17	1, 84	84	0	84	0	84	0	84	0
18	78	78	0	76	-2	78	0	76	-2

Table 3.9: The consensus sequences of motifs discovered by MFACO, footprinting, Gibbs Sampler, BioProspector, and MDGA respectively. (based on the results from Table 3.7 and Table 3.8).

Algorithm		Consensus sequence	Relative entropy
MFACO	BS1		14.417
	BS2		14.408
	BS3		14.399
	BS4		14.398
Footprinting			14.399
Gibbs Sampler			12.75
Prospector			13.618
MDGA			13.92

Minimum Weight Vertex Cover Problem

Contents

4.1 Preliminaries on graphs	68
4.1.1 Graphs	68
4.1.2 The degree of a vertex	69
4.1.3 Graph representations	69
4.2 Minimum Weight Vertex Cover Problem	71
4.2.1 Problem statement	71
4.2.2 Example	72
4.3 MWVCP complexity	73
4.4 Approximation algorithms for MWVCP	73
4.4.1 Bar-Yehuda and Even's algorithm	73
4.4.2 Pitt's randomized algorithm	74
4.4.3 Clarkson's Greedy algorithm	74

Many real-world problems of practical importance can be formulated in terms of graphs. In the following sections, we start with some definitions that pertain to simple graphs in the context of our work and used throughout this

thesis. Then, we describe the minimum weight vertex cover problem. A comprehensive overview on graph theory can be found in [Gross 2003, Bondy 2008, Diestel 2010]

4.1 Preliminaries on graphs

4.1.1 Graphs

A graph G is a pair $G = (V, E)$ of two sets such that $E \subseteq V \times V$; thus, the elements of E are 2-element subsets of V . An element of V is called a *vertex* (also known as *point* or *node*) of the graph G and an element $(u, v) \in E$ is called an *edge* (also known as *line*).

If the pairs $(u, v) \in E$ are ordered pairs, then G is called a *directed graph* (also known as *digraph*), otherwise G is called *undirected* then (u, v) and (v, u) denote the same edge.

Each edge has a set of one or two vertices associated to it, which are called *endpoints*. An edge is said to *join* its endpoints. The endpoints of an edge are said to be *incident* with the edge, and vice versa. Two vertices which are incident with a common edge are *adjacent*. Two distinct adjacent vertices are *neighbours*.

The set of neighbors of a vertex v in a graph G is denoted by $N(v)$ such that $N(v) = \{u | (v, u) \in E \vee (u, v) \in E\}$. An edge with identical endpoints is called a *loop*, and an edge with distinct endpoints a *link*. Two or more links with the same pair of endpoints are said to be *parallel edges*. A *simple graph* is a graph that has no loops or parallel edges. An undirected graph with each pair of its vertices is adjacent is called *complete*. A graph with relatively few possible edges missing is called *dense*, that is, $|E|$ is close to $|V|^2$. On the other hand, a graph with few edges relative to the number of its vertices is called *sparse*, that is, a graph for which $|E|$ is much less than $|V|^2$. Throughout this thesis,

we denote by $n = |V|$ the number of vertices and by $m = |E|$ the number of edges in G .

4.1.2 The degree of a vertex

Let $G = (V, E)$ be a (non-empty) graph. The *degree* (also known as *valency* [Diestel 2010]) of a vertex v in G , denoted by $d(v)$, is the number of edges of G incident with v . In particular, if G is a simple graph, $d(v)$ is the number of neighbors of v in G . A vertex of degree zero is called an *isolated* vertex. Summing all the degrees of vertices in a graph counts the total number of edges twice.

4.1.3 Graph representations

Given a graph $G(V, E)$ whose vertices are numbered $1, 2, \dots, n$ (remember that $n = |V|$) in some arbitrary manner. The two common computational way to represent G are as a collection of *adjacency lists* or as an *adjacency matrix* [Cormen 2009, Levitin 2012]. An example of a graph defined by its adjacency matrix and their adjacency lists is given in Figure 4.1.

4.1.3.1 Adjacency list representation

The adjacency list representation represents the edges of a graph. It consists of one list(set) Adj_v for each vertex v in V . Adj_v contains all the vertices adjacent to v in G (Alternatively, it may contain pointers to these vertices). Further, Adjacency list representation provides a compact way to represent sparse graphs.

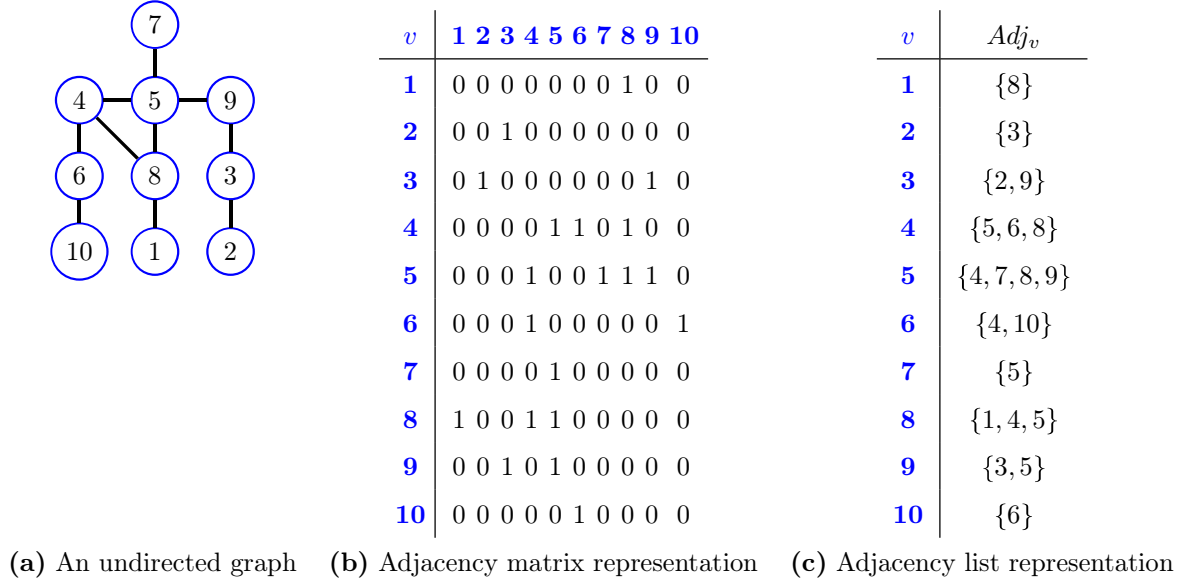


Figure 4.1: Graph representations

4.1.3.2 Adjacency matrix representation

The adjacency matrix representation consists of an $n \times n$ matrix $A = \{a_{ij} \mid i = 1, \dots, n; j = 1, \dots, n\}$ given by

$$a_{ij} = \begin{cases} 1 & \text{if } (i, j) \in E, \\ 0 & \text{otherwise.} \end{cases} \quad (4.1)$$

The element in the i^{th} row and the j^{th} column is equal to 1 if there is an edge from the i^{th} vertex to the j^{th} vertex, and equal to 0 if there is no such edge. The principal advantage of the adjacency matrix representation is that it provides a fast way to determine whether an edge connecting two given vertices. Besides, It seems appropriate to use this form of representation when the graph is dense. Note that the adjacency matrix of an undirected graph is always symmetric.

4.2 Minimum Weight Vertex Cover Problem

4.2.1 Problem statement

A problem instances (G, ω) of the MWVC problem is a tuple that consists of a simple undirected graph $G(V, E)$, where V is the set of vertex and E is the set of edges, and a function $\omega : V \rightarrow \mathbb{R}^+$ that associates a positive weight value $\omega(v)$ to each vertex $v \in V$. The MWVC problem can be formally de-

$$\begin{aligned} & \textbf{minimize} \quad \omega(S) := \sum_{v \in S} \omega(v) \\ & \textbf{subject to} \quad \forall (v_i, v_j) \in E : (v_i \in S) \text{ or } (v_j \in S), \\ & \quad \quad \quad S \subseteq V. \end{aligned}$$

fined as follows:

In the above definition, a *candidate solution* to the problem, denoted by S , is a subset of V . A candidate solution S is a valid solution, a so-called *vertex cover*, if each edge in G has at least one endpoint in S . The objective function value $\omega(S)$ of a candidate solution S is defined as the sum of the weights of the vertices in S . The objective of the MWVC problem is to find a valid candidate solution that minimizes the objective function. We assume that G is represented by means of its adjacency matrix. Therefore, the degree of a vertex $v \in V$ can be easily calculated as follows:

$$d(v) := \sum_{j=1}^n a_{ij} = \sum_{j=1}^n a_{ji} \quad (4.2)$$

Another way to view the MWVCP is as an integer linear programming (ILP) problem. ILP is a mathematical formulation of the problem with a system of linear constraints that can contain both equalities and inequalities, and also a linear objective function that is to be maximized or minimized [Lau 2011]. In our case, we are interested in feasible vectors with all components belonging to the set $\{0, 1\}$. Therefore, an integer variable x_v is associated to each vertex $v \in V$. The values of x_v are restricted to be either 0 or 1 to indicate if v is part of the solution or not. MWVCP can therefore be expressed as follows:

$$\begin{aligned}
&\textbf{minimize} && \sum_{v \in V} \omega(v) \cdot x_v \\
&\textbf{subject to} && x_u + x_v \geq 1 \quad \forall (u, v) \in E, \\
&&& x_v \in \{0, 1\} \quad \forall v \in V.
\end{aligned}$$

The inequality above ensures that the obtained solution contains at least one endpoint of each edge, that is, it represents a valid vertex cover.

4.2.2 Example

In this section we give an illustrative example of MWVCP. As mentioned earlier, a problem instance (G, ω) of MWVCP 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, $|V| = |E| = 10$. The number in each circle denotes the vertex index and its associated weight is given in the right of the figure. For example, the weight of vertex $v = 5$ is $\omega(5) = 25$. The optimal solution is an unordered vertex subset $S^* = \{1, 3, 4, 5, 6\}$, $S^* \subseteq V$, with a total weight of 280.

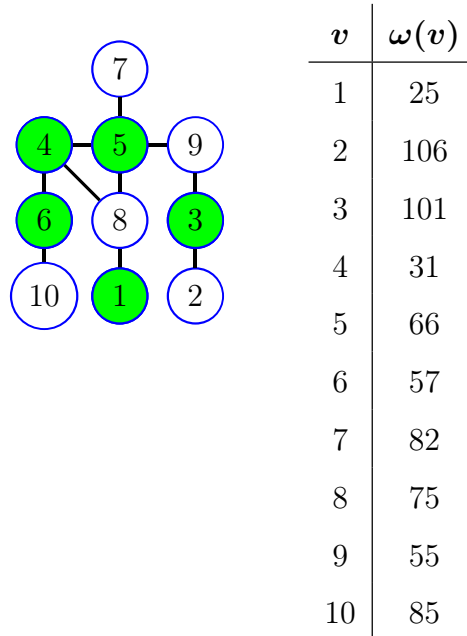


Figure 4.2: An illustrative example of MWVCP

4.3 MWVCP complexity

The MMWVCP is known NP-hard [Karp 1972], so that, there is no polynomial algorithm can achieve the minimum vertex cover unless $P = NP$. Thus, most studies focus on the approximation algorithms for this problem. Dinner and Safra [Dinur 2005] showed that it is \mathcal{NP} -hard to approximate the vertex Cover problem, which is a special case of MWVCP in which all vertices are of unit weight, within any constant factor smaller than $10\sqrt{5}-21$. Moreover, it might be hard to approximate it within an approximation ratio of $2 - \varepsilon$ for all $\varepsilon > 0$ [Khot 2003].

4.4 Approximation algorithms for MWVCP

Greedy algorithms are often used as approximation algorithms to solve optimization problems with a guaranteed performance. With this aim, several greedy algorithms have been suggested in the literature to deal with MWVCP with performance ratio ≤ 2 . In the following, we provide a brief introduction on three of them. A common input to these algorithms is a simple undirected graph $G = (V, E)$ and a weight function $\omega : V \rightarrow \mathbb{R}^+$. A vertex cover $C \subseteq V$ is returned as output.

4.4.1 Bar-Yehuda and Even's algorithm

Bar-Yehuda and Even's algorithm [Bar-Yehuda 1981] selects an edge $(u, v) \in E$, suppose that $w(u) \leq w(v)$, adds u to C and decreases $w(v)$ by $w(u)$. Then, all edges that are incident to u are removed from E . The previous steps are repeated until all edges are removed from E . This approximation algorithm has complexity $O(|V| + |E|)$

4.4.2 Pitt's randomized algorithm

Pitt's randomized algorithm [Pitt 1985] starts by placing all vertices of zero weight in C and arbitrary ordering the edges that remain uncovered. Then, it repeatedly chooses an uncovered edge (u, v) and generates a random number $\alpha \in [0, w(u) + w(v)]$, let assume that $w(u) \leq w(v)$, if $\alpha < w(u)$ then u is added to C , otherwise v is added to C . One can easily see that this is equivalent to placing u in C with a probability $\frac{w(u)}{w(u)+w(v)}$. The algorithm only stops when all edges are treated. Pitt's algorithm has complexity $O(|V| + |E|)$ since each edge is processed once, and an expectation value of ratio ≤ 2 .

4.4.3 Clarkson's Greedy algorithm

Clarkson's algorithm [Clarkson 1983] has complexity $O(|V| \log(|V|) + |E|)$ and an approximation ratio ≤ 2 . It repeatedly chooses a vertex \bar{v} with a minimum value of $\frac{w_c(v)}{d_c(v)}$ over all $v \in V$, then, delete \bar{v} from V and puts it in C . Moreover, all resulting isolated vertices are also removed from V . Note that the current degree $d_c(\cdot)$ is only determined by edges whose both endpoints are not covered. The algorithm continues until all vertices in V are deleted. The resulting set C is output as a vertex cover. Algorithm 6 shows a pseudo-code for this algorithm. It is often the case in applications that actual performance of approximation algorithms is required rather than theoretical time complexity. For this reason, Taoka and Watanabe [Taoka 2012] have implemented the three approximation algorithms (Clarkson's algorithm, Pitt's algorithm and Bar-Yehuda and Even's algorithm) for computing approximate solutions to MWVCP for general graphs. Their performance was compared based on computing results for total 243000 input data consisting of graphs with $100 \leq |V| \leq 3000$, $1 \leq w(v) \leq 10000$. It was observed that Clarkson's algorithm showed the highest capability in producing better solutions in short

computing time.

Algorithm 6 Clarkson's Greedy algorithm

- 1: **input:** A problem instance (G, ω) .
 - 2: $C \leftarrow \emptyset$
 - 3: $V \leftarrow \{v \in V \mid d_c(v) \neq 0\}$
 - 4: **for** each $v \in V$ **do**
 - 5: $w_c(v) \leftarrow w(v)$
 - 6: **while** $V \neq \emptyset$ **do**
 - 7: $\bar{v} \leftarrow \operatorname{argmin} \left\{ \frac{w_c(v)}{d_c(v)} \mid v \in V \right\}$
 - 8: $\varepsilon \leftarrow \frac{w_c(\bar{v})}{d_c(\bar{v})}$
 - 9: **for** each $u \in N(\bar{v})$ (the neighbors of \bar{v}) **do**
 - 10: $w_c(u) \leftarrow w_c(u) - \varepsilon$
 - 11: $C \leftarrow C \cup \{\bar{v}\}$
 - 12: $V \leftarrow V \setminus \{\bar{v}\}$
 - 13: $V \leftarrow \{v \in V \mid d_c(v) \neq 0\}$
 - 14: **output:** C
-

Population Based Iterated Greedy Algorithm for MWVCP

Contents

5.1 Greedy paradigm	76
5.2 Iterated greedy search	79
5.3 Design of a PBIG algorithm for MWVCP	80
5.3.1 Population initialization	83
5.3.2 Construction phase	83
5.3.3 Destruction phase	85
5.3.4 Complexity Considerations	85

In this chapter, we first describe the greedy paradigm and explain its main components. Then, we briefly present a general outline of an iterated greedy algorithm and how it can be adapted to work with a population of solution candidates in parallel. Finally, the details of the novel proposed approach for MWVCP are given.

5.1 Greedy paradigm

The greedy paradigm is one of the fundamental techniques in the design of efficient algorithms, which has yielded exact and approximation algorithms

for numerous optimization problems. A greedy algorithm is a constructive one in the sense that, it builds a solution step by step starting from an empty solution. At each construction step, an item from a finite set of solution components is added to the current partial solution. The item to be added is chosen at each step according to some greedy function, which lends the name to the algorithm. In other words, Each step consists of the following components [Neapolitan 2011]:

- ***A selection procedure*** chooses the next item to add to the set. The selection is performed according to a greedy criterion that satisfies some locally optimal consideration at the time.
- ***A feasibility check*** determines if the new set is feasible by checking whether it is possible to complete this set in such a way as to give a solution to the instance.
- ***A solution check*** determines whether the new set constitutes a solution to the instance,

A typical characteristic of the greedy algorithms is that, once a choice is made on which item to add, this decision is never reconsidered again, that is, without any look-ahead or back-tracking. The main advantages of greedy algorithms are that they are usually simple (easy to understand what they do and how they perform), easy to implement and low time complexity (fast in execution). In contrast, the disadvantage is that the quality of the solutions provided by greedy algorithms is often far from being optimal. Algorithm 7 shows a pseudo-code for a general greedy approach to optimization. Greedy algorithms are typically applied to optimization problems and can be used in at least three ways [Davis 2004]:

- They provide exact algorithms for a variety of problems. Examples of such algorithm include Kruskal's algorithm [Kruskal 1956] for Mini-

Algorithm 7 Pseudo-code for a greedy algorithm

```

1: input:  $C$  is the set of candidate items
2:  $S \leftarrow \emptyset$  // We construct the solution in the set  $S$ 
3: while  $C \neq \emptyset$  and not solution( $S$ ) do
4:    $x \leftarrow \text{select}(C)$ 
5:    $C \leftarrow C \setminus \{x\}$ 
6:   if feasible( $S \cup \{x\}$ ) then
7:      $S \leftarrow S \cup \{x\}$ 
8:   if solution( $S$ ) then
9:     return  $S$ 
10: else
11:   return there are no solutions

```

imum Spanning Tree and Dijkstra's algorithm [Dijkstra 1959] for Shortest Path.

- They are frequently the best approximation algorithms for hard optimization problems.
- Due to their simplicity, they are frequently used as heuristics for hard optimization problems even when their approximation ratios are unknown or known to be poor in the worst-case. In cases where the greedy algorithm does not systematically provide the optimal solution, it is called a greedy heuristic.

In this study, we develop a greedy algorithm as a constructive heuristic search combined with a stochastic improvement technique applied to MWVCP in order to improve the quality of obtaining solutions. This algorithm in turn is used as a construction procedure for the developed population based iterated greedy algorithm.

5.2 Iterated greedy search

Several examples from the literature (see for example [Benedettini 2010, Ruiz 2007, Fanjul-Peyro 2010, Ribas 2011, Lozano 2011]) have shown that constructive heuristics may be enhanced by a simple metaheuristic framework known as an iterated greedy algorithm (IG). An IG algorithm starts with a complete solution. While some termination conditions are not met, it iteratively alternates between two main phases: destruction and construction. During the partial destruction phase some solution components are removed from the incumbent solution. The construction procedure then applies a greedy constructive heuristic to reconstruct a complete candidate solution. Once a candidate solution has been completed, an acceptance criterion decides whether the newly constructed solution will replace the incumbent solution. IG is closely related to Iterated Local Search (ILS) (see Section 3.2): instead of iterating over a local search as done in ILS, IG iterates in an analogous way over construction heuristics. The IG is closely related to iterated local search (ILS). The main difference between the two is that ILS applies local search to perturbations of the current incumbent solution to extend the search space and to escape from deep local optima, whereas in IG the perturbation of the current solution is stronger because it is done by means of the destruction and reconstruction of the solution with a greedy constructive heuristic. Therefore, the IG is better suited than ILS to escape from strong local optima [Ribas 2011]. A general pseudo-code for IG is provided in Algorithm 8. The extension to population-based IG is quite simple. Instead of working on a single incumbent solution, population-based IG maintains at all times a population P of n solutions. The usual IG steps, that is, destruction and reconstruction are, at each iteration, applied to each solution $s \in P$. Adding all solutions generated in this way to P' results in an augmented population P' of n solutions. The population for the next generation is then obtained by removing the worst n solutions

Algorithm 8 Iterated greedy algorithm

```

1:  $s \leftarrow \text{Generate\_Initial\_Solution}()$  // initial solution
2: while termination condition not satisfied do
3:    $s^p \leftarrow \text{Destruction}(s)$ 
4:    $s' \leftarrow \text{Construction}(s^p)$ 
5:    $s \leftarrow \text{Acceptance\_Criterion}(s, s')$ 
6: output:  $\{s_{best} \text{ // best solution found in the search process } \}$ 

```

from both P and P' ($P \cup P'$). The pseudo-code of this procedure is shown in Algorithm 9.

Algorithm 9 Population-based iterated greedy algorithm

```

1:  $P \leftarrow \text{Generate\_Initial\_Population}(n)$  // initial solution
2: while termination condition not satisfied do
3:    $P' \leftarrow \emptyset$ 
4:   for all  $s \in P$  do
5:      $s^p \leftarrow \text{Destruction}(s)$ 
6:      $s' \leftarrow \text{Construction}(s^p)$ 
7:      $P' \leftarrow P' \cup \{s'\}$ 
8:    $P \leftarrow \text{Acceptance\_Criterion}(P, P')$  // Best  $n$  solutions from  $P \cup P'$ 
9: output:  $\{s_{best} \text{ // best solution found in } P\}$ 

```

5.3 Design of a PBIG algorithm for MWVCP

In the following we outline the PBIG (A population-based iterated greedy algorithm) algorithm that we developed for MWVCP. PBIG is a population-based iterated greedy algorithm where a population of solutions, starting from an initial population, is iteratively improved by means of alternating between destruction and reconstruction until termination conditions are met.

Constructive methods are characterized by the fact that they work on partial

solutions. Each constructive method is based on a mechanism for extending partial solutions. Generally, a partial solution S may be extended by adding a solution component from a finite set $\mathcal{N}(S)$ of solution components that are allowed to be added. In the case of MWVCP, $\mathcal{N}(S) \subseteq V \setminus S$ represents the set of vertices that have at least one incident edge which is still uncovered, that is, an edge of which both endpoints are not in S .

Given a (partial) solution S , we henceforth refer to $d_S(v_i) := |\{(v_i, v_j) \in E \mid v_j \notin S\}|$ as the *current degree* of a vertex $v \in V \setminus S$ with respect to S . By this definition $\mathcal{N}(S)$ can technically be defined in the following way:

$$\mathcal{N}(S) := \{v \in V \setminus S \mid d_S(v) \neq 0\} \quad (5.1)$$

Note that when S is a complete solution $\mathcal{N}(S) = \emptyset$. On the other hand, at the start of the algorithm, when S is empty, it holds that $\mathcal{N}(S) = \mathcal{N}_0$ where $\mathcal{N}_0 = \{v \in V \mid d(v) \neq 0\}$, i.e. the isolated vertices are excluded.

A high level description of our algorithm is given in Algorithm 10. Apart from a problem instance (G, ω) , PBIG requires three input parameters to control its flow:

- (i) The population size $pop_size \in \mathbb{Z}^+$.
- (ii) The degree of destruction $p \in [0, 1] \subset \mathbb{R}$, which is a parameter that is used to determine the size of the destruction.
- (iii) The determinism rate $\alpha \in [0, 1] \subset \mathbb{R}$. The latter parameter is used to control the degree of stochasticity of the solution construction procedure.

The algorithm works as follows. First, the pop_size solutions of the initial population are generated by function `GenerateInitialPopulation(pop_size)`. Afterwards, each algorithm iteration consists of the following steps. First, an empty population \mathcal{P}^p called offspring population is created. Then, each solution $S_k \in \mathcal{P}$ ($k = 1, \dots, pop_size$) is partially destroyed using procedure

Algorithm 10 PBIG for the MWVC problem

```

1: input: A problem instance  $(G, \omega)$ , and parameters  $pop\_size \in \mathbb{Z}^+$ ,  $\alpha \in [0, 1] \subset \mathbb{R}$ ,  $p \in [0, 1] \subset \mathbb{R}$ .
2:  $\mathcal{P} \leftarrow \text{GenerateInitialPopulation}(pop\_size)$  // see Algorithm 11
3: while termination condition not satisfied do
4:    $\mathcal{P}^p \leftarrow \emptyset$ 
5:   for each candidate solution  $S_k \in \mathcal{P}$  do
6:      $S_k^p \leftarrow \text{DestroyPartially}(S_k, p)$ 
7:      $S'_k \leftarrow \text{GreedyMWVC}(S_k^p, \alpha)$  // see Algorithm 12
8:      $\mathcal{P}^p \leftarrow \mathcal{P}^p \cup \{S'_k\}$ 
9:    $\mathcal{P} \leftarrow \text{Accept}(\mathcal{P}, \mathcal{P}^p)$ 
10: output:  $\argmin \{\omega(S_k) \mid S_k \in \mathcal{P}, k = 1, \dots, pop\_size\}$ 

```

$\text{DestroyPartially}(S_k, p)$, resulting in a partial solution S_k^p . On the basis of S_k^p , a complete solution S'_k is then constructed using procedure $\text{GreedyMWVC}(S_k^p, \alpha)$. Each newly obtained complete solution is stored in \mathcal{P}^p . Note that the two phases of destruction and construction are applied to each solution independently of each other.

When the iteration is completed, procedure $\text{Accept}(\mathcal{P}, \mathcal{P}^p)$ selects the best pop_size solutions from $\mathcal{P} \cup \mathcal{P}^p$ for the population of the next iteration. This mechanism biases the search process towards the best solutions found during the search, and keeps the population size fixed to pop_size solutions at all times. Finally, the algorithm terminates when either a given limit on the maximum CPU time is reached or a maximum number of iterations has been performed, and the best found solution is returned. The three procedures $\text{GenerateInitialPopulation}(pop_size)$ (see Algorithm 11), $\text{GreedyMWVC}(\cdot, \alpha)$ (see Algorithm 12) and $\text{DestroyPartially}(S_k, p)$ that form the core of PBIG are described in more detail in the following.

5.3.1 Population initialization

The procedure `GenerateInitialPopulation(pop_size)` generates pop_size solutions for the initial population. For the construction of a solution it applies procedure `GreedyMWVC(\cdot, α)` to the empty partial solution $S_0 = \emptyset$. Since this procedure typically includes an element of stochasticity (depending on parameter α), the generated solutions are generally different from each other. The pseudo-code for generating the initial population is shown in Algorithm 11.

Algorithm 11 Procedure `GenerateInitialPopulation(pop_size)`

```

1: input:  $pop\_size$ 
2:  $\mathcal{P} \leftarrow \emptyset$ 
3:  $S_0 \leftarrow \emptyset$ 
4: for  $k = 1, \dots, pop\_size$  do
5:    $S_k \leftarrow \text{GreedyMWVC}(S_0, \alpha)$  // see Algorithm 12
6:    $\mathcal{P} \leftarrow \mathcal{P} \cup \{S_k\}$ 
7: output:  $\mathcal{P} = \{S_1, S_2, \dots, S_{pop\_size}\}$ 

```

5.3.2 Construction phase

The construction phase is assured by the procedure `GreedyMWVC(S, α)`. It takes a partial solution S (which might be empty) as input. At each construction step, the current partial solution S is extended by adding one solution component from $\mathcal{N}(S)$ until a complete solution—that is, a vertex cover—is constructed. 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 $v \in \mathcal{N}(S)$ is rated according to a greedy function $cost(\cdot)$ defined as

$$cost(v) := \omega(v)/d_S(v)^\beta \tag{5.2}$$

When $\beta = 1$ this greedy function is the same as the one used in [Chvátal 1979]. However, experimental tests have shown that better results may sometimes be obtained with $\beta = 2$. Therefore, for each application of the procedure a value for β is randomly adopted from $\{1, 2\}$.

Let v_1^{best} and v_2^{best} ($v_1^{best} \neq v_2^{best}$) be the two vertices from $\mathcal{N}(S)$ with lowest cost values. Moreover, let $cost(v_1^{best}) \leq cost(v_2^{best})$. The determinism rate α is used to decide which one of the two solution components is chosen and added to S . This is done by firstly generating a random value α_0 uniformly distributed over $[0, 1]$. If $\alpha_0 \leq \alpha$ then v_1^{best} is chosen, otherwise v_2^{best} is chosen. The adding of vertices to S stops when $\mathcal{N}(S) = \emptyset$.

At this point solution S may contain vertices whose neighbors—that is, adjacent vertices—belong to S as well. Given a solution S this set may be formally defined as follows:

$$S^u := \{v \in S \mid \mathcal{A}(v) \subseteq S\} \quad (5.3)$$

Where $\mathcal{A}(v)$ is the set of vertices that are adjacent to vertex v in graph G . Note that vertices from S^u do not contribute to the covering and may safely be removed from S . This is done iteratively. While S^u is not empty, one vertex from S^u is chosen as follows. First, each vertex $v \in S^u$ is rated according to a greedy function $rcost(\cdot)$, which is obtained by inverting the greedy function $cost(\cdot)$ from the construction process as explained above (See Eq. 5.4). More specifically,

$$rcost(v) := d_S(v)^\beta / \omega(v). \quad (5.4)$$

Hereby, parameter β adopts the value that was chosen for the construction phase. This iterative process stops once S^u is empty. The pseudo-code of the complete function in Algorithm 12.

5.3.3 Destruction phase

The destruction procedure— $\text{DestroyPartially}(S, p)$ —starts with a complete solution S . Then, $\lfloor p \times |S| \rfloor$ randomly chosen vertices—where $0 < p \leq 1$ —are removed from S in an iterative way. Hereby, at each step exactly one randomly chosen vertex is removed. Parameter p is henceforth called the *degree of destruction*.

5.3.4 Complexity Considerations

The run time complexity of the proposed method is dominated by procedure $\text{GreedyMWVC}(S, \alpha)$ as shown in Algorithm 12. In fact, the complexity of this algorithmic component is the same as the one of the classical greedy heuristic from [Chvátal 1979] for set covering and vertex covering. In other words, the complexity is polynomial in the number of vertices and edges of the input graph. The exact run time complexity depends on the data structures that are used to keep and modify the input graph during the execution of the algorithm. Finally, the run time complexity of one iteration of the proposed PBIG algorithms is pop_size times the run time complexity of procedure $\text{GreedyMWVC}(S, \alpha)$.

Algorithm 12 Procedure GreedyMWVC(S, α)

- 1: **input:** a partial solution S , and parameter α
 - 2: Choose randomly a value for β from $\{1, 2\}$
 - 3: $\mathcal{N}(S) \leftarrow \{v \in V \setminus S \mid d_S(v) \neq 0\}$
 - 4: **while** $\mathcal{N}(S) \neq \emptyset$ **do**
 - 5: $v_1^{best} \leftarrow \operatorname{argmin} \{cost(v) \mid v \in \mathcal{N}(S)\}$
 - 6: $v_2^{best} \leftarrow \operatorname{argmin} \{cost(v) \mid v \in \mathcal{N}(S) \setminus \{v_1^{best}\}\}$
 - 7: $\alpha_0 \leftarrow$ random number uniformly distributed over $[0, 1]$
 - 8: **if** $\alpha_0 \leq \alpha$ **then** $v^{best} \leftarrow v_1^{best}$
 - 9: **else** $v^{best} \leftarrow v_2^{best}$
 - 10: $S \leftarrow S \cup \{v^{best}\}$
 - 11: $\mathcal{N}(S) \leftarrow \{v \in V \setminus S \mid d_S(v) \neq 0\}$
 - 12: $S^u \leftarrow \{v \in S \mid \mathcal{A}(v) \subseteq S\}$
 - 13: **while** $S^u \neq \emptyset$ **do**
 - 14: $v_1^{best} \leftarrow \operatorname{argmax} \{cost(v) \mid v \in S^u\}$
 - 15: $v_2^{best} \leftarrow \operatorname{argmax} \{cost(v) \mid v \in S^u \setminus \{v_1^{best}\}\}$
 - 16: $\alpha_0 \leftarrow$ random number uniformly distributed over $[0, 1]$
 - 17: **if** $\alpha_0 \leq \alpha$ **then** $v^{best} \leftarrow v_1^{best}$
 - 18: **else** $v^{best} \leftarrow v_2^{best}$
 - 19: $S \leftarrow S \setminus \{v^{best}\}$
 - 20: $S^u \leftarrow \{v \in S \mid \mathcal{A}(v) \subseteq S\}$
 - 21: **output:** a complete solution S
-

Experimental Results

Contents

6.1	Problem Instances	88
6.2	Algorithm Tuning	89
6.3	Results and discussions: PBIG versus ACO and ACO+SEE	91
6.3.1	Results Concerning Class SPI	91
6.3.2	Results Concerning Class MPI	94
6.3.3	Results Concerning Class LPI	95
6.3.4	Convergence speed of PBIG	96
6.4	Results and discussions: PBIG versus HSSGA	96
6.4.1	Results Concerning Class SPI	97
6.4.2	Results Concerning Class MPI	97
6.4.3	Results Concerning Class LPI	98

The proposed algorithm was implemented in ANSI C++ using GCC 4.4 for compiling the software, and experimentally evaluated on a cluster of PCs equipped with Intel Xeon X3350 processors with 2667 MHz and 8 Giga-bytes of memory. In the following, we first describe the set of benchmark instances that have been used to test our approach. Then, we report on the tuning process that has been employed for finding well-working settings for the three algorithm parameters. Finally, a comparison of the performance

of PBIG against the performance of recently published algorithms such as ACO [Shyu 2004], HSSGA [Singh 2006], RGES [Balachandar 2009] and ACO-SEED [Jovanovic 2011], is provided.

6.1 Problem Instances

The algorithm was tested on a set of benchmark instances originally proposed in [Shyu 2004]. 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 three different classes:

1. **Class SPI:** the class of *small problem instances* (SPI) contains 400 instances. The number of nodes (n) of instances from this class takes values from $\{10, 15, 20, 25\}$.
2. **Class MPI:** the class of *moderate-size problem instances* (MPI) consists of 710 instances where n takes values from $\{50, 100, 150, 200, 250, 300\}$.
3. **Class LPI:** the class of *large problem instances* (LPI) consists of 15 instances with n from $\{500, 800, 1000\}$.

In each class—and for each n —a whole range of instances exists including rather sparse graphs as well as rather dense graphs.

The instances of the first two classes share the following characteristics:

- 10 problem instances were randomly generated per combination of n and m and results are generally presented as an average over the objective function values obtained for the 10 instances.
- The weight $\omega(v)$ of each vertex $v \in V$ is randomly drawn from a uniform distribution, either from the interval $[20, 120]$ (referred to as Type I) or from the interval $[1, d(v)^2]$ (referred to as Type II), where $d(v)$ is the degree of vertex v .

In contrast, class LPI only consists of one problem instance per combination of n and m and the average quality over 10 different runs of the same algorithm are generally reported as quality measure. The vertex weights are generated similarly as in the case of the Type I instances.

6.2 Algorithm Tuning

As outlined in Chapter 5, our approach has three parameters for which appropriate values must be determined. This concerns (1) the population size (pop_size), (2) the degree of destruction ($p \in (0, 1]$), and (3) the determinism rate ($\alpha \in (0, 1]$). The following values were considered for tuning:

- $pop_size \in \{1, 10, 20, 50, 100\}$;
- $p \in \{0.1, 0.2, 0.3, 0.4, 0.5\}$;
- $\alpha \in \{0.5, 0.6, 0.7, 0.8, 0.9, 1.0\}$, and

All 150 possible combinations of parameter values were considered, that is, a full factorial design approach was applied. For the tuning experiments we selected 406 problem instances for the training set. More precisely, from each instance class we chose for each value of n the instances belonging to the smallest and the largest m -value. Then we applied PBIG with each of the 150 parameter value combinations exactly once to each problem instance from the training set. The computation time limit for each run was set to a maximum number of solution evaluations sol_{max} . As in previous works (see, for example, [Jovanovic 2011]) sol_{max} was set to 20000. For example, in case $pop_size = 50$ the maximum number of iterations for PBIG, denoted by it_{max} , was set to 400. For analyzing the tuning experiments, the rank-based procedure as described in [Benedettini 2010] has been utilized. For each of the 406 problem instances the 150 different parameter settings were

6.3. Results and discussions: PBIG versus ACO and ACO+SEE

ordered based on the solution values they produced for the corresponding instance. In other words, the combination that found the best solution for a certain instance obtained rank 1, and so on. In the case of ties, the same rank was given. Based on these ranks, an average rank for each parameter combination was computed. The combination with the best average rank for the Type I instances of classes SPI and MPI was ($pop_size = 100, p = 0.4, \alpha = 0.5$), whereas the combination with the best average rank concerning the Type II instances of classes SPI and MPI was ($pop_size = 100, p = 0.5, \alpha = 0.8$). Finally, the combination with the best average rank concerning the instances of class LPI was ($pop_size = 100, p = 0.3, \alpha = 0.5$). Therefore, these parameter settings were adopted (for the corresponding instance classes) for all further experiments. In general, the tuning results allowed us to draw the following conclusions: As expected, PBIG performs consistently better when adopting a population of solutions rather than a single solution. In fact, in all cases the best-performing parameter combination used a population size of 100. Second, the degree of destruction should rather drop with growing instance sizes, that is, rather small instances seem to require a higher degree of destruction than larger problem instances. Finally, instances of Type II generally require a stronger degree of determinism during the construction of solutions.

6.3 Results and discussions: PBIG versus ACO and ACO+SEE

The aim of this section is to compare the performance of PBIG with the best available Ant Colony Optimization algorithms from the literature. More precisely, for the comparison we chose the ACO algorithm from [Shyu 2004] and the improved ACO approach (labelled ACO+SEE) from [Jovanovic 2011].

6.3. Results and discussions: PBIG versus ACO and ACO+SEE

ACO and ACO+SEE were applied to all problem instances described in Section 6.1. ACO+SEE can currently be regarded as a state-of-the-art method. In order to be comparable to ACO and ACO+SEE, which were applied for $it_{max} = 2000$ iterations with 10 solution constructions per iteration, we equally used a computation time limit of $sol_{max} = 20000$ solution evaluations for each run of PBIG. Moreover, just like ACO and ACO+SEE, PBIG was applied exactly once to each problem instance from classes SPI and MPI, and 10 times to each problem instance from class LPI.

6.3.1 Results Concerning Class SPI

Tables 6.1 and 6.2 present the numerical results obtained by ACO, ACO+SEE and PBIG for the instances of class SPI. Hereby, Table 6.1 contains the results for the instances of Type I, whereas Table 6.2 contains the results concerning the Type II instances. Remember that class SPI contains 10 different problem instances for each combination of n and m . The values shown in the tables represent the average results obtained for 10 instances. In general, the structure of Tables 6.1 and 6.2 is as follows. The first two columns indicate the number of vertices (n) and the number of edges (m). Column **OPT** refers to the optimal solution values (averaged over 10 instances) as originally provided in [Shyu 2004]. The following two columns provide the average solution quality and the average computation time (in seconds) of ACO. This is the information as given in [Shyu 2004]. The next two columns provide the average solution quality and the average number of solution evaluations of ACO+SEE. Note that the only information provided in [Jovanovic 2011] is the average solution quality of ACO+SEE. We derived the average number of solution evaluations from additional information provided to us by the authors of [Jovanovic 2011]. The results of PBIG are provided in three columns. The first one contains the average solution quality, the second one shows the

6.3. Results and discussions: PBIG versus ACO and ACO+SEED

Table 6.1: Performance of PBIG, ACO and ACO+SEED on instances of class SPI (Type I).

n	m	OPT	ACO		ACO+SEE		PBIG			Stat.
			Avg	Time (s)	Avg	Evals	Avg	Time (s)	Evals	Sign.
10	10	284.0	284.0	0.000	284.0	10	284.0	0.000	4.1	\ominus
	20	398.7	398.7	0.008	398.7	18	398.7	0.000	17.5	\ominus
	30	431.3	431.3	0.003	431.3	12	431.3	0.000	2.7	\ominus
	40	508.5	508.5	0.003	508.5	12	508.5	0.000	1.4	\ominus
15	20	441.9	441.9	0.005	441.9	147	441.9	0.000	1.3	\ominus
	40	570.4	574.2	0.011	570.4	230	570.4	0.000	2.5	\ominus
	60	726.2	729.0	0.008	726.2	45	726.2	0.000	2.8	\ominus
	80	807.5	814.6	0.010	807.5	25	807.5	0.000	3.2	\ominus
	100	880.0	880.0	0.008	880.0	14	880.0	0.000	3.1	\ominus
20	20	473.0	473.0	0.005	473.0	30	473.0	0.000	3.3	\ominus
	40	659.3	661.4	0.016	660.3	435	659.3	0.000	22.9	\ominus
	60	861.8	861.8	0.014	861.8	189	861.8	0.000	7.7	\ominus
	80	898.8*	905.4	0.016	899.9	13	898.0	0.000	17.1	\ominus
	100	1026.2	1026.8	0.016	1026.2	538	1026.2	0.000	2.8	\ominus
	120	1038.2	1041.5	0.017	1038.2	1011	1038.2	0.000	5.2	\ominus
25	40	756.6	756.6	0.019	756.6	103	756.6	0.000	2.1	\ominus
	80	1008.1	1009.6	0.022	1008.1	638	1008.1	0.000	12.9	\ominus
	100	1106.9	1107.4	0.025	1109.1	1155	1106.9	0.000	12.0	\ominus
	150	1264.0	1264.0	0.031	1264.0	50	1264.0	0.000	21.0	\ominus
	200	1373.4	1377.7	0.030	1373.4	84	1373.4	0.000	2.7	\ominus
AVG			777.4	0.013	776.0	238.0	775.7	0.000	7.42	

* We believe that 898.8 is a mistake from [Shyu 2004]. The 10 solutions of PBIG resulting in a value of 898.0) were thoroughly checked and are given in Appendix A.

average computation time (in seconds), and the third one provides the average number of solution evaluations. In the case of PBIG we provide both the average computation time and the average number of solution evaluations in order to be able to compare to both ACO and ACO+SEE. Finally, the last

6.3. Results and discussions: PBIG versus ACO and ACO+SEE 93

table column gives information about the outcome of a statistical test. More specifically, for each combination of n and m we applied the Mann-Whitney test [Corder 2009] to the results of PBIG and ACO+SEE, which is a current state-of-the-art algorithm. Symbol \ominus indicates that there is no statistical difference between the results of the two algorithms. On the other side, symbol \oplus indicates that the results of PBIG are better than those of ACO+SEE in a statistically significant way (using a standard significance level of 0.05). As a final remark, note that in each table row the result of the best algorithm in the comparison is given in bold font. The last table row provides an average over the whole table. Concerning the interpretation of the results we can observe that PBIG always generates the optimal result. This can be confirmed by comparing the values in column **OPT** with the average solution qualities obtained by PBIG. In contrast, ACO+SEE is only able to generate the optimal results in 32 out of 40 cases, and ACO is only able to provide the optimal result in 16 out of 40 cases. However, as can be seen in the last column of Tables 6.1 and 6.2, there is no statistical difference between the results of ACO+SEE and PBIG. For what concerns computation time, we can observe that PBIG is very fast. In fact, computation times are smaller than 0.001 seconds in nearly all cases. ACO, on the contrary, has average computation times of 0.013 seconds (see the last row of Table 6.1) and 0.015 seconds (see the last row of Table 6.2). In order to be able to compare these computation times we must note that ACO has been executed on a computer with an AMD processor of 1700 MHz and 256 MB of RAM. Therefore, we can assume that the machines that we used for executing PBIG are about twice as fast as the one used for executing ACO. When comparing the average number of solution evaluations of ACO+SEE and PBIG, we can observe that PBIG needs between 30 times (Table 6.1) and 40 times (Table 6.2) less solution evaluations than ACO+SEE. In summary, the small problem instances of class SPI do not pose any challenge to PBIG.

6.3.2 Results Concerning Class MPI

The results for the problem instances of class MPI are presented in Tables 6.3 and 6.4, in the same way as described in the previous section for the case of instance class SPI. The additional column **RGES** in Table 6.4 presents the results of the RGES algorithm [Balachandar 2009] which was only applied to the Type II instances of class MPI. Note also that optimal solutions are not known for the instances of this class, due to their increased size. Therefore, column **OPT** is missing. Again we can observe that PBIG is superior to the competitor algorithms. In all 70 cases PBIG obtains the best result of the comparison. Altogether, ACO, RGES, and ACO+SEE are only able to match the results of PBIG in two cases. Concerning statistical significance, it can be observed that for rather small problem instances of this class, the improvement of PBIG over ACO+SEE is still not statistically significant. On the other side, for all larger problem instances, the improvement of PBIG over ACO+SEE is statistically significant. Altogether, statistical significance could be determined in 49 out of 70 cases. For what concerns a comparison of computation time, we can observe that PBIG needs about twice as much computation time than ACO. This is taking into consideration the fact that the machines used to execute PBIG are about twice as fast as the machine used to run ACO. However, given that the stopping criterion for ACO and PBIG has been the same, this only means that ACO gets stuck earlier during the search process, while PBIG is able to make a better use of the allotted computation time. Finally, comparing the number of solution evaluations needed by ACO+SEE and PBIG, it can be observed that PBIG obtains a better solution quality than ACO+SEE using, on average, about 3-4 times fewer solution evaluation.

6.3.3 Results Concerning Class LPI

Table 6.5 presents a comparison of PBIG with ACO+SEE for what concerns the instances of class LPI. Remember that the instances of this class are the largest ones of the whole benchmark set. In this case we only compare to ACO+SEE, which is the best available algorithm for these instances. In contrast to instance classes SPI and MPI, class LPI only contains one single instance per combination of n and m . Therefore, the results are given as averages over 10 runs per instance. For each of the two algorithms, columns **Best**, **Avg** and **Evals** provide, respectively, the value of the best solution found, the average solution quality over 10 runs and the average number of solution evaluations needed for reaching the best solutions in 10 runs. In addition, in the case of PBIG we provide the average computation time in column **Time (s)**. For each combination we compare between the best performance (columns **Best**) and the average performance (columns **Avg**) of PBIG and ACO+SEE. The better among the two is always indicated in bold font. Concerning the comparison, we can observe that PBIG outperforms ACO+SEE both for what concerns the best solutions found and the average solution quality obtained. The only exception is case $(n = 500, m = 5000)$ where ACO+SEE finds a better solution than PBIG. It is interesting to note that in 13 out of 15 cases the average solution quality obtained by PBIG is better than the value of the best solution found by ACO+SEE. Moreover, the last table column indicates that in all cases PBIG improves over ACO+SEE in a statistical significant way. Concerning the number of solution evaluations, PBIG and ACO+SEE make use of, on average, a comparable number of solution evaluations.

6.3.4 Convergence speed of PBIG

A convergence analysis presented in [Jovanovic 2011] for cases $(n = 500, m = 2000)$ (see row 3 of Table 6.5), $(n = 800, m = 10000)$ (see row 10 of Table 6.5)

and $(n = 1000, n = 1000)$ (see row 11, Table 6.5) (see Figure 6.1) has shown that ACO+SEE converges after 200-500 iterations, which corresponds to 2000-5000 solution evaluations. The rest of the computation time is not utilized by ACO+SEE. From the latter figure we can see that in spite of the superiority of ACO+SEED over ACO, both are more susceptible to converge to local optima especially for large problem instances. Figure 6.2 through Figure 6.4 show such a convergence analysis also for PBIG. Each graphic shows the evolution of the value of the best solution found by each of the 10 applications of PBIG over time. First, we can observe that—in all three cases—after very few iterations PBIG is able to find better solutions than the best ones found by ACO+SEE. Moreover, PBIG does not stagnate. Good solutions can also be found towards the end of a run.

6.4 Results and discussions: PBIG versus HSSGA

The aim of this section is to compare the performance of PBIG with a hybrid heuristic based steady-state genetic algorithm (HSSGA) developed by In Singh and Gupta [Singh 2006] for MWVCP. The genetic algorithm generates vertex covers which are then tuned by a heuristic. The heuristic first select the set of those vertices in the vertex cover whose edges are fully covered by other vertices within the cover and then it consecutively deletes such vertices from the vertex cover until the set is empty. It was tested on all problem instances described in Section 6.1. In the same manner as for PBIG, HSSGA were applied for $it_{max} = 2000$ iterations (generations) with a population size of 100 individuals and was applied exactly once to each problem instance from classes SPI and MPI, and 10 times for each problem instance from class LPI. HSSGA was executed on a 2.4GHz Pentium 4 processor-based system with 512 MB RAM.

6.4.1 Results Concerning Class SPI

Tables 6.6 and 6.7 compare the performance of PBIG with HSSGA on problem instances of class SPI. The data for HSSGA are taken from [Singh 2006]. In these tables, the average solution quality and the average computation time (in seconds) are provided for each approach. We can observe that both approaches are able to find the optimal solutions to all instances of class SPI and that in a negligible computation time. In fact, they achieve equal performance for this test case.

6.4.2 Results Concerning Class MPI

Tables 6.8 and 6.9 compare the performance of PBIG with HSSGA on problem instances of class MPI. We should note that PBIG performs significantly better than HSSGA, especially when the size of the problem instance increases. PBIG outperforms HSSGA in 10 out 30 cases and in 13 out 14 cases for problem instances of Type I and Type II respectively. There were also no significant differences in running time if we take in consideration the number of evaluated solutions needed to converge to a good quality solution.

6.4.3 Results Concerning Class LPI

Table 6.10 shows a comparison of PBIG with HSSGA for what concerns the instances of class LPI. PBIG outperforms ACO+SEE in term of the average solution quality obtained. It can be seen that PBIG outperforms HSSGA in 11 out 14 cases and with equal performance in one case (for $n = 500$, $m = 800$). When comparing the average computation time of ACO+SEE and PBIG (see see row 16, Table 6.10), we can observe that PBIG needs more time to find its best solution than HSSGA. It does not mean that the latter is faster. This may be an explanation for the performance improvement of PBIG as the quality of the solution usually keeps improving until the last

iterations. On the other hand, HSSGA may get trapped in local optima and converge rapidly after a certain number of iterations. Moreover, an interesting observation from the experimental results is that the relative difference in the solution quality between PBIG and the other algorithms increases with an increase of the problem instance. That is, PBIG achieve better performance as compared to ACO, ACO+SEED and HSSGA especially when the size of the problem is large.

Table 6.2: Performance of PBIG, ACO and ACO+SEED on instances of class SPI (Type II).

n	m	ACO		ACO+SEED		PBIG		Stat.	
		Avg	Time (s)	Avg	Evals	Avg	Time (s)	Evals	Sign.
10	10	18.8	0.003	18.8	45	18.8	0.000	2.4	\ominus
	20	51.1	0.003	51.1	118	51.1	0.000	2.6	\ominus
	30	127.9	0.003	127.9	10	127.9	0.000	1.7	\ominus
	40	268.3	0.010	268.3	20	268.3	0.000	2.0	\ominus
15	20	34.7	0.005	34.7	108	34.7	0.000	14.0	\ominus
	40	170.5	0.010	170.5	716	170.5	0.000	3.3	\ominus
	60	360.5	0.008	360.5	27	360.5	0.000	1.5	\ominus
	80	697.9	0.014	697.9	1663	697.9	0.000	3.9	\ominus
	100	1130.4	0.008	1130.4	1292	1130.4	0.001	62.4	\ominus
20	20	32.9	0.011	32.9	997	32.9	0.000	1.9	\ominus
	40	111.6	0.017	111.8	1337	111.6	0.000	30.6	\ominus
	60	254.1	0.016	254.1	265	254.1	0.000	12.3	\ominus
	80	452.2	0.016	452.3	235	452.2	0.000	9.1	\ominus
	100	775.2	0.016	775.2	50	775.2	0.000	2.7	\ominus
25	120	1123.1	0.017	1123.1	456	1123.1	0.001	23.4	\ominus
	40	98.7	0.025	98.7	313	98.7	0.000	30.8	\ominus
	80	372.7	0.026	373.0	238	372.7	0.000	19.6	\ominus
	100	595.0	0.028	595.1	443	595.0	0.000	4.2	\ominus
	150	1289.9	0.030	1290.9	106	1289.9	0.000	4.8	\ominus
AVG	200	2709.5	0.030	2709.5	1850	2709.5	0.000	7.9	\ominus
		534.7	0.015	533.8	514.5	533.8	0.000	12.1	

Table 6.3: Performance of PBIG, ACO and ACO+SEED on instances of class MPI (Type I).

n	m	ACO		ACO+SEE		PBIG			Stat.
		Avg	Time (s)	Avg	Evals	Avg	Time (s)	Evals	Sign.
50	50	1282.1	0.063	1280.9	414	1280.0	0.016	1152.0	\ominus
	100	1741.1	0.083	1740.7	2258	1735.3	0.007	205.9	\ominus
	250	2287.4	0.097	2280.6	325	2272.3	0.006	66.4	\ominus
	500	2679.0	0.102	2669.3	331	2661.9	0.003	29.9	\ominus
	750	2959.0	0.125	2957.3	329	2951.0	0.027	61.1	\ominus
	1000	3211.2	0.117	3199.8	2291	3193.7	0.019	24.4	\ominus
100	100	2552.9	0.273	2544.0	3722	2537.6	0.019	502.8	\ominus
	250	3626.4	0.367	3614.9	1049	3602.7	0.057	428.5	\ominus
	500	4692.1	0.433	4636.4	1577	4600.6	0.182	703.0	\oplus
	750	5076.4	0.502	5082.8	1454	5045.5	0.088	253.4	\oplus
	1000	5534.1	0.456	5522.7	1808	5509.4	0.084	156.3	\ominus
	2000	6095.7	0.589	6068.3	448	6051.9	0.501	307.0	\ominus
150	150	3684.9	0.691	3676.8	3834	3667.3	0.088	1000.5	\oplus
	250	4769.7	0.891	4754.9	2714	4720.3	0.116	713.6	\oplus
	500	6224.0	1.194	6228.7	5862	6165.7	0.294	869.4	\oplus
	750	7014.7	1.042	6996.3	1705	6963.7	0.522	1348.4	\oplus
	1000	7441.8	1.206	7383.6	2090	7368.8	0.536	930.7	\oplus
	2000	8631.2	1.103	8597.2	1441	8562.0	0.824	400.3	\ominus
200	3000	8950.2	0.966	8940.2	584	8899.8	1.300	346.4	\oplus
	250	5588.7	1.674	5572.4	5335	5551.9	0.157	812.2	\ominus
	500	7259.2	2.160	7233.7	3930	7192.4	0.547	1374.4	\oplus
	750	8349.8	2.602	8300.3	3651	8274.5	0.664	995.6	\oplus
	1000	9262.2	2.221	9208.4	3178	9150.6	1.019	1084.6	\oplus
	2000	10916.5	2.437	10891.1	2770	10831.0	2.726	1629.2	\oplus
250	3000	11689.1	2.497	11680.4	1937	11600.2	2.866	690.6	\oplus
	250	6197.8	2.273	6169.2	8418	6148.7	0.392	1956.0	\oplus
	500	8538.8	4.016	8495.9	8871	8440.7	1.334	4114.7	\oplus
	750	9869.4	4.047	9815.5	8428	9752.8	2.447	4423.5	\oplus
	1000	10866.6	3.755	10791.0	6190	10753.7	2.371	2138.9	\oplus
	2000	12917.7	3.942	12827.0	6229	12757.6	3.471	1229.8	\oplus
300	3000	13882.5	4.276	13830.6	5023	13723.5	3.233	896.9	\oplus
	5000	14801.8	3.842	14735.9	1507	14676.7	22.508	3004.5	\oplus
	300	7342.7	4.322	7326.6	7370	7296.0	0.711	2557.3	\oplus
	500	9517.4	5.178	9491.9	7273	9403.1	1.596	4051.3	\oplus
	750	11166.9	6.055	11156.5	8729	11038.1	3.349	3816.9	\oplus
	1000	12241.7	6.231	12163.7	9922	12108.9	3.095	2325.1	\oplus
AVG	2000	14894.9	6.488	14834.6	4976	14749.9	10.982	3366.3	\oplus
	3000	16054.1	6.299	15910.5	3284	15848.2	11.636	2821.3	\oplus
	5000	17545.4	6.558	17479.8	3584	17350.6	17.283	1450.5	\oplus
	AVG	7881.0	2.338	7848.5	3713.9	7806.1	2.489	1390.8	

Table 6.4: Performance of PBIG, ACO and ACO+SEED on instances of class MPI (Type II).

n	m	ACO		RGES	ACO+SEE		PBIG			Stat.
		Avg	Time (s)	Avg	Avg	Evals	Avg	Time (s)	Evals	Sign.
50	50	83.9	0.072	83.9	83.9	1299	83.7	0.002	109.7	\ominus
	100	276.2	0.097	276.2	274.4	2716	271.2	0.003	81.3	\ominus
	250	1886.8	0.111	1886.4	1870.3	2528	1853.4	0.010	81.1	\ominus
	500	7915.9	0.120	7914.5	7876.7	914	7825.1	0.009	45.1	\oplus
	750	20134.1	0.111	20134.1	20087.6	505	20079.0	0.010	23.0	\ominus
100	50	67.4	0.184	67.4	67.2	266	67.2	0.002	66.0	\ominus
	100	169.1	0.334	169.1	167.8	2244	166.6	0.017	199.6	\ominus
	250	901.7	0.514	890.4	895.3	6630	886.5	0.065	377.8	\oplus
	500	3726.7	0.481	3725.3	3707.0	3806	3693.6	0.101	213.3	\oplus
	750	8754.5	0.444	8745.5	8742.3	3000	8680.2	0.129	235.4	\oplus
150	50	65.8	0.292	65.8	65.9	190	65.8	0.001	28.1	\ominus
	100	144.7	0.583	144.7	144.1	1491	144.0	0.026	231.3	\ominus
	250	625.7	1.387	624.4	624.8	6546	616.0	0.187	601.4	\oplus
	500	2375.0	1.908	2365.2	2358.6	5925	2331.5	0.572	850.6	\oplus
	750	5799.2	1.295	5798.6	5707.0	4928	5698.7	0.550	518.0	\oplus
200	50	59.6	0.463	59.6	59.6	48	59.6	0.001	26.6	\ominus
	100	134.7	0.981	132.6	134.6	3634	134.5	0.021	188.9	\ominus
	250	488.7	2.413	488.4	487.9	7303	483.1	0.280	1054.5	\oplus
	500	1843.6	3.423	1843.6	1818.7	10850	1804.3	1.286	1524.8	\oplus
	750	4112.8	3.600	4112.8	4077.0	8833	4043.6	0.768	766.0	\oplus
250	250	423.2	3.311	423.2	421.2	8553	419.0	0.476	1127.4	\oplus
	500	1457.4	5.781	1457.4	1454.3	12500	1435.7	4.219	4113.3	\oplus
	750	3315.9	5.983	3315.9	3289.4	7865	3261.0	2.935	1693.0	\oplus
	1000	6058.2	6.297	6058.2	6040.0	11586	5989.4	7.978	3223.0	\oplus
	2000	26149.1	4.859	26149.1	25932.1	5145	25658.5	11.809	1967.0	\oplus
	5000	171917.2	4.856	171917.2	171500.7	3402	170269.1	37.770	1624.1	\oplus
300	250	403.9	5.372	403.9	402.7	7713	399.5	0.353	1109.1	\oplus
	500	1239.1	9.155	1239.1	1237.3	11971	1216.4	5.439	4510.6	\oplus
	750	2678.2	10.994	2678.2	2674.1	9491	2639.4	6.545	4135.8	\oplus
	1000	4895.5	9.045	4895.5	4867.9	11492	4796.3	15.204	4941.9	\oplus
	2000	21295.2	7.242	21295.2	21107.7	11399	20891.6	10.911	2084.1	\oplus
	5000	143243.5	6.553	143243.5	142292.6	3388	141265.3	27.674	1080.4	\oplus
AVG		13832.6	3.071	13831.4	13764.7	5567.5	13663.4	4.230	1213.5	

Table 6.5: Performance of PBIG, ACO and ACO+SEED on instances class LPI.

n	m	ACO+SEE			PBIG			Stat.	
		Best	Avg	Evals	Best	Avg	Time (s)	Evals	Sign.
500	500	12675	12687.7	10935	12616	12620.0	1.601	3962.8	⊕
	1000	16516	16574.9	10794	16465	16470.1	10.240	12661.1	⊕
	2000	21000	21093.0	7700	20863	20870.8	9.283	4855.1	⊕
	5000	27294	27585.5	6143	27318	27428.2	34.707	6656.1	⊕
	10000	29573	29796.4	6602	29573	29666.8	36.405	2620.4	⊕
800	500	15049	15069.9	4282	15025	15025.0	2.535	5275.9	⊕
	1000	22792	22852.1	10860	22747	22763.0	11.589	9172.0	⊕
	2000	31680	31786.9	8397	31355	31422.6	58.008	17795.6	⊕
	5000	38830	38906.7	11485	38665	38718.7	113.842	12737.6	⊕
	10000	44499	44691.7	6747	44396	44397.8	96.467	4521.9	⊕
1000	1000	24856	24925.4	16377	24746	24763.1	14.435	10034.7	⊕
	5000	45446	45588.7	10225	45255	45295.4	178.311	14860.3	⊕
	10000	51875	52105.0	10298	51378	51540.9	325.956	12113.2	⊕
	15000	58394	58654.8	7277	58014	58145.2	363.179	7906.7	⊕
	20000	60010	60268.2	6896	59790	59847.9	647.563	9291.2	⊕
AVG		33365.9	33505.8	9001.2	33213.7	33265.0	126.941	8964.3	

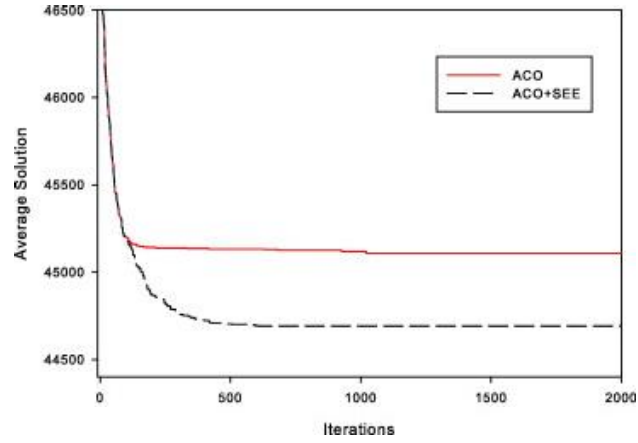
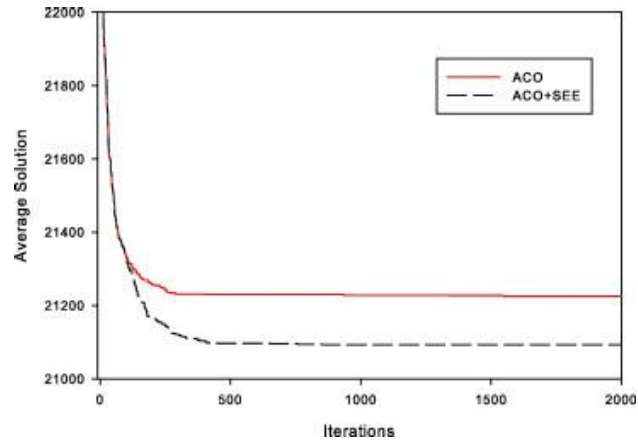
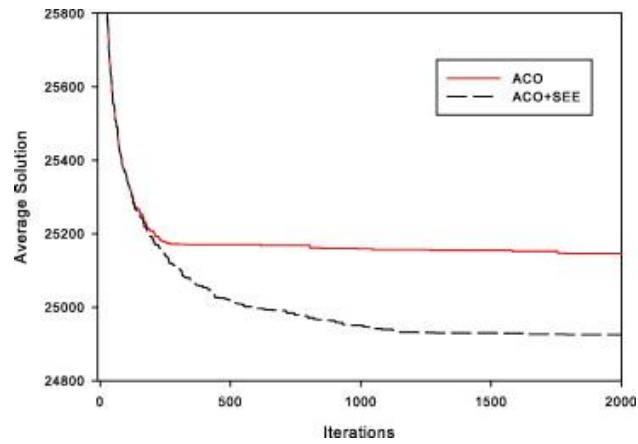
(a) Case ($n = 500, m = 2000$).(b) Case ($n = 800, m = 10000$).(c) Case ($n = 1000, m = 1000$).

Figure 6.1: Convergence speed of ACO and ACO+SEED for the test case of three problem instances of class LPI [Jovanovic 2011].

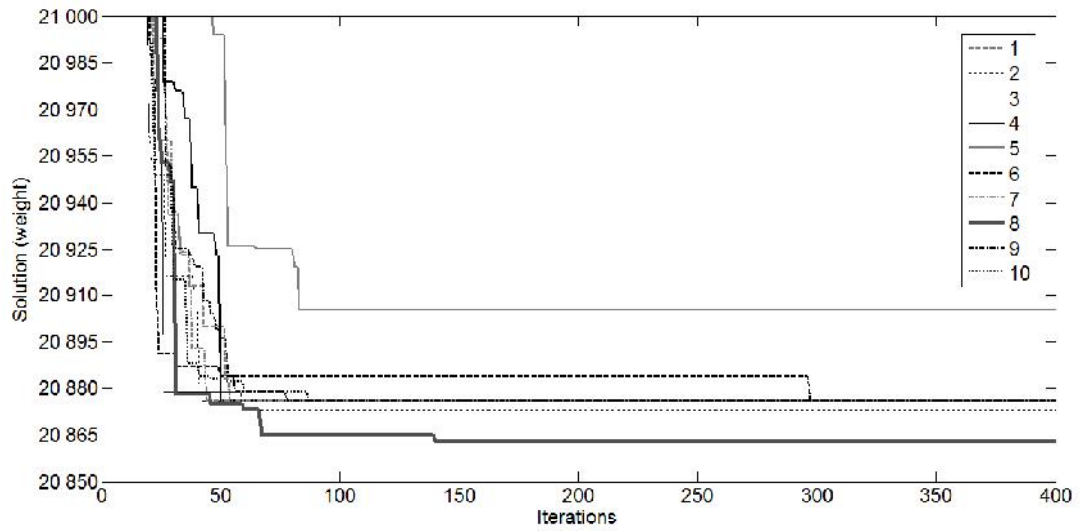


Figure 6.2: Convergence speed for problem instance $n = 500, m = 2000$

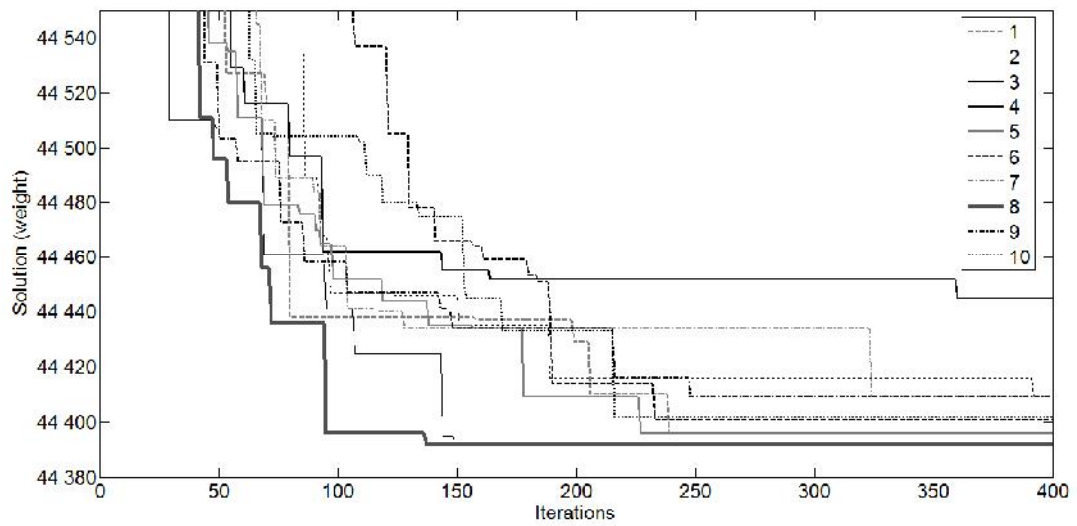


Figure 6.3: Convergence speed for problem instance $n = 800, m = 10000$

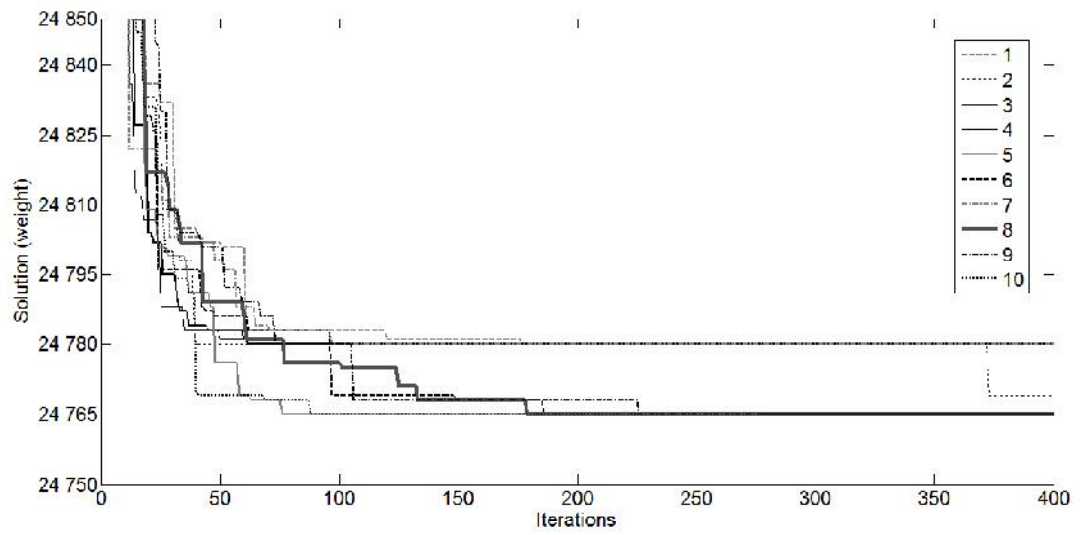


Figure 6.4: Convergence speed for problem instance $n = 1000$, $m = 1000$

Table 6.6: Performance of PBIG and HSSGA on instances of class SPI (Type I).

n	m	OPT	HSSGA		PBIG	
			Avg	Time (s)	Avg	Time (s)
10	10	284.0	284.0	0.000	284.0	0.000
	20	398.7	398.7	0.000	398.7	0.000
	30	431.3	431.3	0.000	431.3	0.000
	40	508.5	508.5	0.000	508.5	0.000
15	20	441.9	441.9	0.000	441.9	0.000
	40	570.4	570.4	0.000	570.4	0.000
	60	726.2	726.2	0.000	726.2	0.000
	80	807.5	807.5	0.000	807.5	0.000
	100	880.0	880.0	0.000	880.0	0.000
20	20	473.0	473.0	0.000	473.0	0.000
	40	659.3	659.3	0.000	659.3	0.000
	60	861.8	861.8	0.000	861.8	0.000
	80	898.8	898.0	0.001	898.0	0.000
	100	1026.2	1026.2	0.001	1026.2	0.000
	120	1038.2	1038.2	0.000	1038.2	0.000
25	40	756.6	756.6	0.000	756.6	0.000
	80	1008.1	1008.1	0.000	1008.1	0.000
	100	1106.9	1106.9	0.000	1106.9	0.000
	150	1264.0	1264.0	0.000	1264.0	0.000
	200	1373.4	1373.4	0.000	1373.4	0.000
AVG			775.7	0.0001	775,7	0

Table 6.7: Performance of PBIG and HSSGA on instances of class SPI (Type II).

n	m	OPT	HSSGA		PBIG	
			Avg	Time (s)	Avg	Time (s)
10	10	18.8	18.8	0.000	18.8	0.000
	20	51.1	51.1	0.000	51.1	0.000
	30	127.9	127.9	0.000	127.9	0.000
	40	268.3	268.3	0.000	268.3	0.000
15	20	34.7	34.7	0.000	34.7	0.000
	40	170.5	170.5	0.000	170.5	0.000
	60	360.5	360.5	0.000	360.5	0.000
	80	697.9	697.9	0.000	697.9	0.000
	100	1130.4	1130.4	0.000	1130.4	0.001
20	20	32.9	32.9	0.001	32.9	0.000
	40	111.6	111.6	0.000	111.6	0.000
	60	254.1	254.1	0.000	254.1	0.000
	80	452.2	452.2	0.000	452.2	0.000
	100	775.2	775.2	0.000	775.2	0.000
	120	1123.1	1123.1	0.000	1123.1	0.001
25	40	98.7	98.7	0.000	98.7	0.000
	80	372.7	372.7	0.000	372.7	0.000
	100	595.0	595.0	0.000	595.0	0.000
	150	1289.9	1289.9	0.000	1289.9	0.000
25	200	2709.5	2709.5	0.000	2709.5	0.000
AVG			533.8	0.00005	533.8	0.0001

Table 6.8: Performance of PBIG and HSSGA on instances of class MPI (Type I).

n	m	HSSGA		PBIG	
		Avg	Time (s)	Avg	Time (s)
50	50	1280.0	0.002	1280.0	0.016
	100	1735.3	0.003	1735.3	0.007
	250	2272.3	0.003	2272.3	0.006
	500	2661.9	0.002	2661.9	0.003
	750	2951.0	0.003	2951.0	0.027
	1000	3194.4	0.000	3193.7	0.019
100	100	2534.2	0.025	2537.6	0.019
	250	3602.7	0.040	3602.7	0.057
	500	4600.6	0.136	4600.6	0.182
	750	5045.5	0.029	5045.5	0.088
	1000	5508.2	0.026	5509.4	0.084
	2000	6051.9	0.036	6051.9	0.501
150	150	3666.9	0.082	3667.3	0.088
	250	4721.1	0.145	4720.3	0.116
	500	6172.7	0.398	6165.7	0.294
	750	6965.0	0.219	6963.7	0.522
	1000	7370.4	0.223	7368.3	0.536
	2000	8549.4	0.298	8562.0	0.824
	3000	8899.8	0.105	8899.3	1.300
200	250	5551.9	0.316	5551.3	0.157
	500	7202.3	0.247	7192.4	0.547
	750	8273.3	0.304	8274.5	0.664
	1000	9153.4	0.248	9150.6	1.019
	2000	10836.0	0.969	10831.0	2.726
	3000	11602.4	0.374	11600.2	2.866
250	250	6151.6	0.440	6148.7	0.392
	500	8442.9	2.602	8440.7	1.334
	750	9772.9	1.957	9752.8	2.447
	1000	10760.1	1.019	10753.7	2.371
	2000	12755.5	1.426	12757.6	3.471
	3000	13738.0	0.559	13723.5	3.233
	5000	14671.3	0.793	14676.7	22.508
300	300	7295.8	2.143	7296.0	0.711
	500	9416.8	1.415	9403.1	1.596
	750	11041.0	4.003	11038.1	3.349
	1000	12116.3	2.055	12108.9	3.095
	2000	14744.8	3.873	14749.9	10.982
	3000	15846.3	1.175	15848.2	11.636
300	5000	17366.3	1.423	17350.6	17.283
AVG		7808.3	0.747	7806.1	2.489

Table 6.9: Performance of PBIG and HSSGA on instances of class MPI (Type II).

n	m	HSSGA		PBIG	
		Avg	Time (s)	Avg	Time (s)
50	50	83.7	0.002	83.7	0.002
	100	271.2	0.003	271.2	0.003
	250	1853.4	0.001	1853.4	0.010
	500	7825.1	0.002	7825.1	0.009
	750	20079.0	0.000	20079.0	0.010
100	50	67.2	0.013	67.2	0.002
	100	166.6	0.019	166.6	0.017
	250	886.5	0.059	886.5	0.065
	500	3693.6	0.022	3693.6	0.101
	750	8680.2	0.022	8680.2	0.129
150	50	65.8	0.025	65.8	0.001
	100	144.1	0.049	144.0	0.026
	250	616.0	0.073	616.0	0.187
	500	2331.5	0.379	2331.5	0.572
	750	5702.3	0.117	5698.7	0.550
200	50	59.6	0.042	59.6	0.001
	100	134.5	0.085	134.5	0.021
	250	483.1	0.336	483.1	0.280
	500	1804.3	0.273	1804.3	1.286
	750	4046.5	0.545	4043.6	0.768
250	250	419.1	0.657	419.0	0.476
	500	1436.8	0.384	1435.7	4.219
	750	3265.2	0.373	3261.0	2.935
	1000	5993.3	0.457	5989.4	7.978
	2000	25701.2	0.579	25658.5	11.809
	5000	170306.3	0.856	170269.1	37.770
300	250	399.6	0.481	399.5	0.353
	500	1216.9	1.559	1216.4	5.439
	750	2644.5	1.111	2639.4	6.545
	1000	4808.7	2.264	4796.3	15.204
	2000	20936.1	0.788	20891.6	10.911
	5000	141278.3	1.646	141265.3	27.674
AVG		13668.8	0.413	13663.4	4.230

Table 6.10: Performance of PBIG and HSSGA on instances of class LPI.

n	m	HSSGA		PBIG	
		Avg	Time (s)	Avg	Time (s)
500	500	12621.2	14.256	12620.0	1.601
	1000	16470.4	10.331	16470.1	10.240
	2000	20882.6	7.223	20870.8	9.283
	5000	27393.4	10.266	27428.2	34.707
	10000	29627.2	8.553	29666.8	36.405
800	500	15025.0	60.296	15025.0	2.535
	1000	22774.5	96.205	22763.0	11.589
	2000	31433.8	91.797	31422.6	58.008
	5000	38742.9	58.266	38718.7	113.842
	10000	44478.3	34.753	44397.8	96.467
1000	1000	24730.2	138.868	24763.1	14.435
	5000	45347.2	62.514	45295.4	178.311
	10000	51736.0	102.958	51540.9	325.956
	15000	58203.9	63.946	58145.2	363.179
	20000	59948.0	37.700	59847.9	647.563
AVG		33294.3	53.195	33265.0	126.941

Conclusions and Future Works

Contents

7.1	Conclusions	111
7.2	Future Works	113

7.1 Conclusions

This thesis describes an investigation into the development of stochastic optimization technique to the minimum weight vertex cover problem (MWVCP). MWVCP is a fundamental NP-complete problem in graph theory. One of the noticeable problems usually encountered by most of the previous works on MWVCP is that their performance decrease significantly relative to the problem size. They seem to not scale well particularly for large problem instances. This work has been motivated by an interest in developing a new optimization approach that tackles this problem in a more effective way than currently exists.

On the other hand, several recent studies from the literature have shown that constructive heuristics may be enhanced by a simple and effective metaheuristic framework known as an iterated greedy algorithm (IG). IG is a stochastic local search algorithm recently developed for combinatorial optimization problems, which has exhibited state-of-the-art performances for a considerable number of problems. It generates a sequence of solutions by iterating over

greedy constructive heuristics using destruction and reconstruction phases. During the destruction phase, some solution components are removed from a previously constructed complete candidate solution. The reconstruction phase is then driven to build the resulting partial solution in order to obtain again a complete solution. An acceptance criterion is used to decide whether the newly constructed solution should replace the current one. Moreover, scalability for high-dimensional problems has become an essential requirement for modern optimization techniques.

Additionally, Algorithms based on a population of potential solutions, rather than a single solution at a time, can in effect search many local optima and, thereby, increase the likelihood of finding the global optima. Therefore, they are also less susceptible to converge to local optima.

Taking the previous considerations into account, we have proposed a population-based iterated greedy (PBIG) approach to tackle the MWVCP and capable of reaching high-quality solutions especially for large size problem instances. The proposed algorithm maintains a population of solutions and applies, at each iteration, the basic steps of an iterated greedy algorithm to each member of the population. The best solutions among the current population and the newly generated solutions are accepted for the population of the subsequent iteration.

The destructive procedure simply consists in randomly removing a certain fraction of the vertices from the incumbent solution. For the construction phase, we develop a new randomized greedy construction heuristic based on the heuristic described in [Chvátal 1979, Clarkson 1983]. Randomized greedy heuristic have an important advantage over deterministic greedy one in that it has the potential for generating multiple local optimal solutions.

Tuning experiments have confirmed that two important points. First, working with a population of solutions rather than being restricted to a single solution can greatly avoid the rapid convergence to local optima and increase

diversity. The version of PBIG using a population size of 100 performs better than versions of PBIG using less solutions per iteration. Second, the degree of destruction should rather drop with growing instance sizes, that is, small instances seem to require a higher degree of destruction than larger problem instances.

Our computational results based on 1125 benchmark instances of different sizes reveal that PBIG performs slightly better than the currently available algorithms from the literature, especially when the instance size grows, such as ACO [Shyu 2004], HSSGA [Singh 2006], RGES [Balachandar 2009] and ACO+SEED [Jovanovic 2011] and that not only in solution quality but also in computation time (respectively, number of solution evaluations). Furthermore, PBIG is able to solve all small problem instances to optimality, in negligible computation time.

7.2 Future Works

In the future, several issues could be further investigated to improve the performance of PBIG:

- Search for some greedy problem-specific heuristics, which can be exploited in a more advanced construction phase.
- Extend our approach by using several greedy heuristics at the same time, rather than being restricted to a single one.
- As both GRASP (see Section 3.3) and PBIG apply a greedy randomized construction procedure to generate feasible solution. However, GRASP makes use of local search for refining the generated solutions to local optima. So, The performance of PBIG might be improved by incorporating a suitable local search strategy. Following this idea, simulated annealing (SA) can be employed as a local search as done in a recent

work by Voß and Fink [Voß 2012] where the latter approach is hybridized with a reactive tabu search (TS) to incorporate key features of both SI and TS to solve the same problem.

We believe that PBIG may be also successfully extended to other NP-complete problems as well especially those that are similar to the MWVCP such as the Minimum Weight Dominating Set Problem [Potluri 2013] and the Minimum Weighted Set Cover Problem [Korte 2012].

Minimum Weighted Dominating Set. Given an undirected graph $G = (V, E)$, a dominating set is a subset $V^* \subseteq V$ such that each $v \in V$ is either a member of V^* or there is a vertex $u \in V^*$ for which $(u, v) \in E$. Given a weight function $\omega : V \rightarrow \mathbb{R}^+$, the minimum weighted dominating set is a dominating set V^* such that $\sum_{v \in V^*} \omega(v)$ is minimized.

Minimum Weighted Set Cover. Let \mathcal{U} be a set of elements and \mathcal{S} be a collection of non-empty subsets of \mathcal{U} . Given a weight function $\omega : \mathcal{S} \rightarrow \mathbb{R}^+$, the minimum weighted set cover is a subset $\mathcal{S}^* \subseteq \mathcal{S}$ such that $\sum_{S \in \mathcal{S}^*} \omega(S)$ is minimized and $\bigcup_{S \in \mathcal{S}^*} S = \mathcal{U}$, that is, \mathcal{S}^* covers \mathcal{U} . MWVCP is equivalent to the special case of this problem, where each set contains exactly two elements.

Best solutions of case

$(n = 20, m = 80)$

Here we report the best solutions found by PBIG for the 10 instances of case $(n = 20, m = 80)$ from Table 6.1. Note that the 20 vertices are numbered from 1 to 20.

Instance no. 1

$$S = \{6, 9, 15, 5, 3, 8, 7, 17, 13, 18, 16, 11, 12, 20\}$$

$$\omega(S) = 899$$

Instance no. 2

$$S = \{9, 8, 15, 13, 6, 2, 14, 5, 3, 12, 10, 7, 18, 1, 20, 16\}$$

$$\omega(S) = 947$$

Instance no. 3

$$S = \{14, 10, 4, 16, 19, 8, 7, 18, 17, 5, 20, 12, 11, 2\}$$

$$\omega(S) = 866$$

Instance no. 4

$$S = \{3, 15, 14, 4, 17, 5, 7, 11, 1, 19, 10, 20, 9, 2\}$$

$$\omega(S) = 768$$

Instance no. 5

$$S = \{16, 17, 1, 20, 6, 15, 2, 3, 7, 18, 9, 11, 14, 4\}$$

$$\omega(S) = 976$$

Instance no. 6

$$S = \{20, 13, 7, 1, 14, 15, 16, 2, 9, 12, 10, 11, 5, 3\}$$

$$\omega(S) = 905$$

Instance no. 7

$$S = \{7, 11, 4, 18, 9, 8, 3, 14, 20, 12, 1, 16, 10, 19\}$$

$$\omega(S) = 944$$

Instance no. 8

$$S = \{18, 15, 13, 16, 9, 14, 19, 8, 5, 4, 2, 6, 10, 17, 3\}$$

$$\omega(S) = 899$$

Instance no. 9

$$S = \{15, 9, 4, 18, 3, 11, 20, 5, 1, 14, 10, 6, 2, 13\}$$

$$\omega(S) = 970$$

Instance no. 10

$$S = \{18, 15, 3, 13, 1, 9, 4, 6, 12, 10, 5, 7, 20, 2\}$$

$$\omega(S) = 806$$

Average solution quality: 898.0

Bibliography

- [Affenzeller 2005] M. Affenzeller, S. Wagner and S. Winkler. *GA-Selection Revisited from an ES-Driven Point of View*. In J. Mira and J. R. Àlvarez, editors, Artificial Intelligence and Knowledge Engineering Applications: A Bioinspired Approach, volume 3562 of *Lecture Notes in Computer Science*, pages 262–271. Springer Berlin Heidelberg, 2005. (Cited on page [39](#).)
- [Avazbeigi 2009] M. Avazbeigi. *An Overview of Complexity Theory*. In R. Farahani and M. Hekmatfar, editors, Facility Location: Concepts, Models, Algorithms and Case Studies, pages 19–36. Springer, 2009. (Cited on pages [15](#) and [18](#).)
- [Balachandar 2009] S. R. Balachandar and K. Kannan. *A meta-heuristic algorithm for vertex covering problem based on gravity*. International Journal of Mathematical and Statistical Sciences, vol. 1, no. 3, pages 130–136, 2009. (Cited on pages [5](#), [7](#), [88](#), [94](#) and [113](#).)
- [Bar-Yehuda 1981] R. Bar-Yehuda and S. Even. *A linear-time approximation algorithm for the weighted vertex cover problem*. Journal of Algorithms, vol. 2, no. 2, pages 198 – 203, 1981. (Cited on page [73](#).)
- [Bazaraa 2006] M. Bazaraa, H. Sherali and C. Shetty. Nonlinear Programming: Theory And Algorithms. Wiley-Interscience, 2006. (Cited on page [22](#).)
- [Benedettini 2010] S. Benedettini, C. Blum and A. Roli. *A Randomized Iterated Greedy Algorithm for the Founder Sequence Reconstruction Problem*. In C. Blum and R. Battiti, editors, Learning and Intelligent Optimization, 4th International Conference (LION 4), volume 6073

- of *Lecture Notes in Computer Science*, pages 37–51. Springer, 2010. (Cited on pages 6, 79 and 90.)
- [Blum 2003] C. Blum and A. Roli. *Metaheuristics in combinatorial optimization: Overview and conceptual comparison*. ACM Comput. Surv., vol. 35, no. 3, pages 268–308, 2003. (Cited on pages 25 and 27.)
- [Blum 2008] C. Blum and D. Merkle. *Swarm intelligence*. Springer, 2008. (Cited on pages 41 and 42.)
- [Bonabeau 2001] E. Bonabeau and C. Meyer. *Swarm intelligence: A whole new way to think about business*. Harvard business review, vol. 79, no. 5, pages 106–115, 2001. (Cited on page 41.)
- [Bondy 2008] J. A. Bondy and U. S. R. Murty. Graph theory, volume 244 of *Graduate Texts in Mathematics*. Springer, 2008. (Cited on page 68.)
- [Booker 1987] L. Booker. *Improving search in genetic algorithms*. In Genetic Algorithms and Simulated Annealing, pages 61–73. Morgan Kaufmann Publisher, Inc., 1987. (Cited on page 32.)
- [Che 2005] D. Che, Y. Song and K. Rasheed. *MDGA: motif discovery using a genetic algorithm*. In Proceedings of the 2005 conference on Genetic and evolutionary computation, GECCO '05, pages 447–452. ACM, 2005. (Cited on pages vi, 8, 46, 47, 57, 59, 60 and 64.)
- [Chinneck 2012] J. W. Chinneck. Practical optimization: A gentle introduction. 2012. <http://www.sce.carleton.ca/faculty/chinneck/po.html>. (Cited on page 2.)
- [Chvátal 1979] V. Chvátal. *A greedy heuristic for the set-covering problem*. Mathematics of Operations Research, vol. 4, no. 3, pages 233–235, 1979. (Cited on pages 4, 6, 84, 85 and 112.)

- [Clarkson 1983] K. L. Clarkson. *A Modification of the Greedy Algorithm for Vertex Cover*. Information Processing Letters, vol. 16, no. 1, pages 23–25, 1983. (Cited on pages 6, 74 and 112.)
- [Corder 2009] G. W. Corder and D. I. Foreman. *Comparing two unrelated samples: the Mann-Whitney U-test*, chapter in: Nonparametric statistics for Non-Statisticians: A Step-by-Step approach, pages 57–78. John Wiley & Sons, Hoboken, New Jersey, USA, 2009. (Cited on page 93.)
- [Cormen 2009] T. H. Cormen, C. E. Leiserson, R. L. Rivest and C. Stein. Introduction to algorithms. MIT Press, Massachusetts, USA, 3 édition, 2009. (Cited on pages 16, 18, 20 and 69.)
- [Das 2007] M. K. Das and H. K. Dai. *A survey of the DNA motif finding algorithms*. BMC Bioinformatics, vol. 8 (Supplement 7), no. S21, 2007. (Cited on page 46.)
- [Davis 2004] S. Davis and R. Impagliazzo. *Models of greedy algorithms for graph problems*. In Proceedings of the fifteenth annual ACM-SIAM symposium on Discrete algorithms, SODA '04, pages 381–390, Philadelphia, PA, USA, 2004. Society for Industrial and Applied Mathematics. (Cited on page 77.)
- [Diestel 2010] R. Diestel. Graph theory. Graduate Texts in Mathematics. Springer, 4 édition, 2010. (Cited on pages 68 and 69.)
- [Dijkstra 1959] E. W. Dijkstra. *A note on two problems in connexion with graphs*. Numerische Mathematik, vol. 1, pages 269–271, 1959. (Cited on page 78.)
- [Dinur 2005] I. Dinur and S. Safra. *On the hardness of approximating minimum vertex-cover*. Annals of Mathematics, vol. 162, no. 1, pages 439–485, 2005. (Cited on pages 4 and 73.)

- [Dorigo 1992] M. Dorigo. *Optimization, learning and natural algorithms*. Ph. D. Thesis, Politecnico di Milano, Italy, 1992. (Cited on page 41.)
- [Dorigo 1996] M. Dorigo, V. Maniezzo and A. Coloni. *The Ant System: Optimization by a colony of cooperating agents*. IEEE Transactions on Systems, Man, and Cybernetics - Part B, vol. 26, no. 1, pages 29–41, 1996. (Cited on pages 7, 41, 44 and 57.)
- [Dorigo 2003] M. Dorigo and T. Stützle. *The ant colony optimization metaheuristic: Algorithms, applications, and advances*. In F. Glover and G. Kochenberger, editors, Handbook of metaheuristics, pages 251–285. Kluwer Academic Publishers, 2003. (Cited on page 42.)
- [Dorigo 2004] M. Dorigo and T. Stützle. Ant Colony Optimization. MIT Press, Cambridge, 2004. (Cited on pages 7, 23, 25, 27, 41 and 42.)
- [Dorigo 2006] M. Dorigo, M. Birattari and T. Stützle. *Ant colony optimization*. Computational Intelligence Magazine, IEEE, vol. 1, no. 4, pages 28–39, 2006. (Cited on page 42.)
- [Edelkamp 2012] S. Edelkamp and S. Schrödl. Heuristic search: theory and applications. Morgan Kaufmann Publishers, MA, USA, 2012. (Cited on page 15.)
- [El-Ghazali 2009] T. El-Ghazali. Metaheuristics: From design to implementation. Wiley Publishing, 2009. (Cited on pages 1, 23, 25, 28, 33 and 34.)
- [Fanjul-Peyro 2010] L. Fanjul-Peyro and R. Ruiz. *Iterated greedy local search methods for unrelated parallel machine scheduling*. European Journal of Operational Research, vol. 207, no. 1, pages 55–69, 2010. (Cited on pages 6 and 79.)

- [Feo 1995] T. Feo and M. Resende. *Greedy Randomized Adaptive Search Procedures*. Journal of Global Optimization, vol. 6, pages 109–133, 1995. (Cited on page 36.)
- [Freitas 2002] A. A. Freitas. Data mining and knowledge discovery with evolutionary algorithms. Springer-Verlag, New York, 2002. (Cited on page 39.)
- [Garey 1979] M. Garey and D. Johnson. Computers and intractability: A guide to the theory of NP-Completeness. W. H. Freeman, 1979. (Cited on page 18.)
- [Gen 2000] M. Gen and R. Cheng. Genetic Algorithms and Engineering Optimization. John Wiley & Sons, 2000. (Cited on pages 32, 38 and 40.)
- [Gill 1982] P. Gill, W. Murray and M. Wright. Practical optimization. Emerald Group Publishing Limited, 1982. (Cited on page 3.)
- [Glover 1986] F. Glover. *Future paths for integer programming and links to artificial intelligence*. Computers & Operations Research, vol. 13, no. 5, pages 533–549, 1986. (Cited on page 27.)
- [Glover 1997] F. Glover and M. Laguna. Tabu search. Kluwer Academic Publishers, Norwell, MA, USA, 1997. (Cited on page 27.)
- [Goldberg 1989] D. E. Goldberg. Genetic algorithms in search, optimization, and machine learning. Addison-Wesley, 1989. (Cited on pages 38 and 39.)
- [Gross 2003] J. L. Gross and J. Yellen. Handbook of graph theory. CRC Press, Florida, USA, 2003. (Cited on page 68.)

- [Hoos 2004] H. H. Hoos and T. Stützle. *Stochastic local search: Foundations & applications*. Elsevier / Morgan Kaufmann, San Francisco (CA), USA, 2004. (Cited on pages 6, 20 and 33.)
- [Jones 2004] N. C. Jones and P. A. Pevzner. *An Introduction to Bioinformatics Algorithms*. MIT Press, Cambridge, 2004. (Cited on pages 47 and 50.)
- [Jovanovic 2011] R. Jovanovic and M. Tuba. *An ant colony optimization algorithm with improved pheromone correction strategy for the minimum weight vertex cover problem*. *Applied Soft Computing*, vol. 11, no. 8, pages 5360–5366, 2011. (Cited on pages v, 5, 7, 88, 89, 91, 92, 93, 96, 103 and 113.)
- [Karp 1972] R. M. Karp. *Reducibility among combinatorial problems*. In R. E. Miller and J. W. Thatcher, editors, *Complexity of Computer Computations*, pages 85–103, New York, 1972. Plenum Press. (Cited on pages 4, 18 and 73.)
- [Karpenko 2005] O. Karpenko, J. Shi and Y. Dai. *Prediction of MHC class II binders using the ant colony search strategy*. *Artificial Intelligence in Medicine*, vol. 35, no. 1, pages 147–156, 2005. (Cited on page 46.)
- [Kaya 2009] M. Kaya. *MOGAMOD: Multi-objective genetic algorithm for motif discovery*. *Expert Systems with Applications*, vol. 36, no. 2, pages 1039–1047, 2009. (Cited on page 46.)
- [Keith 2002] J. M. Keith, P. Adams, D. Bryant, D. P. Kroese, K. R. Mitchelson, D. Cochran and G. H. Lala. *A simulated annealing algorithm for finding consensus sequences*. *Bioinformatics*, vol. 18, no. 11, pages 1494–1499, 2002. (Cited on page 46.)

- [Khot 2003] S. Khot and O. Regev. *Vertex Cover Might be Hard to Approximate to within $2 - \varepsilon$* . In 18th Annual IEEE Conference on Computational Complexity (CCC'03), pages 379–386, 2003. (Cited on pages 4 and 73.)
- [Khuri 1994] S. Khuri and T. Bäck. *An evolutionary heuristic for the minimum vertex cover problem*. In J. Hopf, editor, Genetic Algorithms within the Framework of Evolutionary Computation, KI-94 Workshop, Saarbrücken, Germany, pages 86–90, 1994. (Cited on page 4.)
- [Korte 2012] B. Korte and J. Vygen. Combinatorial optimization: Theory and algorithms, volume 21 of *Algorithms and Combinatorics*. Springer, 5 édition, 2012. (Cited on pages 16, 18 and 114.)
- [Kruskal 1956] J. B. Kruskal. *On the Shortest Spanning Subtree of a Graph and the Traveling Salesman Problem*. Proceedings of the American Mathematical Society, vol. 7, no. 1, pages 48–50, 1956. (Cited on page 77.)
- [Lau 2011] L. C. Lau, R. Ravi and M. Singh. Iterative methods in combinatorial optimization. Cambridge University Press, New York, USA, 2011. (Cited on page 71.)
- [Lawrence 1993] C. Lawrence, J. Liu S. Altschul M. Boguski, A. Neuwald and J. Wootton. *A Gibbs sampling strategy for multiple alignments*. Science, vol. 262, no. 5131, pages 208–214, 1993. (Cited on pages 7, 46 and 51.)
- [LcBailey 1994] T. LcBailey and C. Elkan. *Fitting a mixture model by expectation maximization to discover motifs in biopolymers*. In Proc. of the 2nd Int. Conf. on Intelligent Systems for Molecular Biology, pages

- 28–36, Menlo Park, California, 1994. AAAI Press. (Cited on pages 8 and 57.)
- [Levitin 2012] A. Levitin. Introduction to the design & analysis of algorithms. Pearson Education, 3 édition, 2012. (Cited on pages 16 and 69.)
- [Liao 2003] Y. J. Liao, C. B. Yang and S. H. Shiau. *Motif finding in biological sequences*. In Proc. of 2003 Symposium on Digital Life and Internet Technologies, pages 89–98, Tainan, Taiwan, 2003. (Cited on page 46.)
- [Liu 2001] X. Liu, D. L. Brutlag and J. S. Liu. *BioProspector: Discovering conserved DNA motifs in upstream regulatory regions of co-expressed genes*. In Pac. Symp. Biocomput, pages 127–138, 2001. (Cited on pages 8, 51 and 57.)
- [Liu 2004] F. Liu, J. Tsai, R. Chen, S. Chen and S. Shih. *FMGA: Finding Motifs by Genetic Algorithm*. In Proceedings of the 4th IEEE Symposium on Bioinformatics and Bioengineering, BIBE '04, pages 459–466, Washington, DC, USA, 2004. IEEE Computer Society. (Cited on pages 8, 46, 57 and 58.)
- [Lourenço 2003] H. Lourenço and O. Martinand T. Stützle. *Iterated Local Search*. In F. Glover and G. Kochenberger, editors, Handbook of Metaheuristics, pages 321–353. KLUWER ACADEMIC PUBLISHERS, 2003. (Cited on page 33.)
- [Lozano 2011] M. Lozano, D. Molina and C. García-Martínez. *Iterated Greedy for the Maximum Diversity Problem*. European Journal of Operational Research, vol. 214, no. 1, pages 31–38, 2011. (Cited on pages 6 and 79.)
- [Neapolitan 2011] R. Neapolitan and K. Naimipour. Foundations of Algorithms. Jones & Bartlett Publishers, USA, 4 édition, 2011. (Cited on page 77.)

- [Osman 1996] I. Osman and G. Laporte. *Metaheuristics: A bibliography*. *Annals of Operations Research*, vol. 63, no. 5, pages 511–623, 1996. (Cited on page 27.)
- [Paquete 2002] L. Paquete and T. Stützle. *An Experimental Investigation of Iterated Local Search for Coloring Graphs*. In S. Cagnoni, J. Gottlieb, E. Hart, M. Middendorf and G. Raidl, editors, *Applications of Evolutionary Computing*, volume 2279 of *Lecture Notes in Computer Science*, pages 122–131. Springer, 2002. (Cited on page 34.)
- [Parsopoulos 2010] K. E. Parsopoulos and M. N. Vrahatis. *Particle swarm optimization and intelligence: Advances and applications*. IGI Global, Hershey, USA, 2010. (Cited on page 1.)
- [Pevzner 2000] P. A. Pevzner and S. Sze. *Combinatorial approaches to finding subtle signals in DNA sequences*. In *Proc. of the 8th Int. Conf. on Intelligent Systems for Molecular Biology (ISMB'00)*, pages 269–278, San Diego, California, 2000. AAAI Press. (Cited on page 46.)
- [Pitt 1985] L. Pitt. *A simple probabilistic Approximation Algorithm for Vertex Cover*. Rapport technique YaleU/DCS/TR-404, Department of Computer Science, Yale University, 1985. (Cited on pages 4 and 74.)
- [Potluri 2013] A. Potluri and A. Singh. *Hybrid metaheuristic algorithms for minimum weight dominating set*. *Applied Soft Computing*, vol. 13, no. 1, pages 76–88, 2013. (Cited on page 114.)
- [Price 2003] A. Price, S. Ramabhadran and P. A. Pevzner. *Finding subtle motifs by branching from sample strings*. *Bioinformatics*, vol. 19, no. suppl 2, pages ii149–ii155, 2003. (Cited on page 48.)

- [Reeves 1993] C. Reeves, editor. Modern heuristic techniques for combinatorial problems. John Wiley & Sons, Inc., New York, NY, USA, 1993. (Cited on page 25.)
- [Ribas 2011] I. Ribas, R. Companys and X. Tort-Martorell. *An iterated greedy algorithm for the flowshop scheduling problem with blocking*. Omega, vol. 39, no. 3, pages 293–301, 2011. (Cited on pages 6 and 79.)
- [Rothlauf 2011] F. Rothlauf. Design of modern heuristics. Natural Computing Series. Springer, 2011. (Cited on pages 15 and 26.)
- [Ruiz 2007] R. Ruiz and T. Stützle. *A simple and effective iterated greedy algorithm for the permutation flowshop scheduling problem*. European Journal of Operational Research, vol. 177, no. 3, pages 2033–2049, 2007. (Cited on pages 6 and 79.)
- [Schrijver 2003] A. Schrijver. Combinatorial optimization: Polyhedra and efficiency, volume 24 of *Algorithms and Combinatorics*. Springer, 2003. (Cited on page 13.)
- [Seehuus 2005] R. Seehuus, A. Tveit and O. Eidsberg. *Discovering biological motifs with genetic programming*. In Proceedings of the 2005 conference on Genetic and evolutionary computation, GECCO '05, pages 401–408. ACM, 2005. (Cited on page 46.)
- [Shyu 2004] S. Shyu, P. Yin and B. Lin. *An Ant Colony Optimization Algorithm for the Minimum Weight Vertex Cover Problem*. Annals of Operations Research, vol. 131, no. 1–4, pages 283–304, 2004. (Cited on pages 5, 7, 88, 91, 92 and 113.)
- [Singh 2006] A. Singh and A. K. Gupta. *A hybrid heuristic for the minimum weight vertex cover problem*. Asia-Pacific Journal of Operational Re-

- search, vol. 23, no. 2, pages 273â–285, 2006. (Cited on pages [5](#), [7](#), [88](#), [96](#), [97](#) and [113](#).)
- [Spall 2003] J. C. Spall. Introduction to stochastic search and optimization. John Wiley & Sons, Hoboken, New Jersey, USA, 2003. (Cited on page [3](#).)
- [Stormo 1989] G. D. Stormo and G. W. Hartzell. *Identifying protein-binding sites from unaligned DNA fragments*. Proc. Natl. Acad. Sci., vol. 86, no. 4, pages 1183–1187, 1989. (Cited on pages [47](#), [57](#) and [59](#).)
- [Stützle 2000] T. Stützle and H. Hoos. *MAX-MIN ant system*. Future Generation Computer Systems, vol. 16, no. 9, pages 889–914, 2000. (Cited on pages [7](#), [45](#), [46](#) and [57](#).)
- [Taoka 2012] S. Taoka and T. Watanabe. *Performance comparison of approximation algorithms for the minimum weight vertex cover problem*. In 2012 IEEE International Symposium on Circuits and Systems (IS-CAS), pages 632–635. IEEE, 2012. (Cited on page [74](#).)
- [Thijs 2001] G. Thijs, M. Lescot, K. Marchal, S. Rombauts, B. De Moor, P. Rouz and Y. Moreau. *A higher order background model improves the detection of regulatory elements by Gibbs Sampling*. Bioinformatics, vol. 17, no. 12, pages 1113–1122, 2001. (Cited on pages [8](#), [51](#), [55](#) and [57](#).)
- [Tomba 2005] M. Tomba and et al. *Assessing computational tools for the discovery of transcription factor binding sites*. Nature Biotechnology, vol. 23, no. 1, pages 137–144, 2005. (Cited on page [46](#).)
- [Voß 2012] S. Voß and A. Fink. *A hybridized tabu search approach for the minimum weight vertex cover problem*. Journal of Heuristics, vol. 18, no. 6, pages 869–876, 2012. (Cited on pages [5](#) and [114](#).)

- [Wang 2010] H. Wang and S. Wang. *GRASP for Low Autocorrelated Binary Sequences*. In D. S. Huang and *al.*, editors, Advanced Intelligent Computing Theories and Applications, volume 6215 of *Lecture Notes in Computer Science*, pages 252–257. Springer Berlin Heidelberg, 2010. (Cited on page 37.)
- [Yang 2010] X. Yang. Engineering optimization: An introduction with meta-heuristic applications. John Wiley & Sons, Inc., Hoboken, New Jersey, 2010. (Cited on pages 12 and 25.)

DESIGN OF A LEARNING METHOD FOR AUTOMATIC DATA EXTRACTION

Abstract

Optimization is a scientific discipline that is concerned with the extraction of optimal solutions for a problem, among alternatives. Many challenging applications in business, economics, and engineering can be formulated as optimization problems. However, they are often complex and difficult to solve by an exact method within a reasonable amount of time.

In this thesis we propose a novel approach called *population based iterated greedy algorithm* in order to efficiently explore and exploit the search space of one of the NP-hard combinatorial optimization problems namely the *minimum weigh vertex cover problem*. It is a fundamental graph problem with many important real-life applications such as, for example, in wireless communication, circuit design and network flows.

An extensive experimental evaluation on a commonly used set of benchmark instances shows that our algorithm outperforms current state-of-the-art methods not only in solution quality but also in computation time.

Keywords: stochastic algorithms, combinatorial optimization problem, minimum weight vertex cover problem, greedy heuristic

CONCEPTION D'UNE METHODE AUTOMATIQUE D'APPRENTISSAGE POUR LA FOUILLE DE DONNEES

Résumé

L'optimisation est une discipline scientifique qui s'intéresse à l'extraction des solutions optimales pour un problème donné, présentant plusieurs solutions. De nombreuses applications stimulantes dans l'industrie, en finances et en ingénierie peuvent être formulées comme des problèmes d'optimisation. Cependant, ils sont souvent complexes et difficiles à résoudre avec exactitude dans des temps de calcul acceptables.

Dans cette thèse, nous proposons une nouvelle approche appelée *algorithme glouton itératif à base de population* pour explorer et exploiter d'une manière adéquate l'espace de recherche de l'un des problèmes d'optimisation combinatoires de classe NP-difficile. Il s'agit du *problème de la couverture par les sommets de poids minimum* qui est un problème fondamental dans la théorie de graphe, couvrant de nombreuses applications dans les domaines de la communication sans fils, de la conception des circuits et du contrôle du flux dans les réseaux.

Une évaluation expérimentale approfondie sur les différents jeux de données disponibles dans la littérature montre que l'algorithme développé, concurrence aisément en terme de qualité des solutions et des temps de calcul associés, les méthodes de pointe actuelles dans la plupart des cas.

Mots-clés: algorithmes stochastiques, optimisation combinatoire, problème de la couverture par les sommets de poids minimum, heuristique glouton.