UNIVERSITAT POLITÈCNICA DE CATALUNYA (UPC)

Algorithms for the Weighted Independent Domination Problem

Author:
Guillem Rodríguez Corominas

Director:
Maria J. Blesa
Computer Science Department (CS), UPC

Co-director:
Christian Blum
IIIA-CSIC, Bellaterra

Bachelor Degree in Informatics Engineering
Computing Specialization
Barcelona School of Informatics (FIB)

July 4, 2019

Abstract

The weighted independent domination problem is an NP-hard combinatorial optimization problem in graphs. This problem has only been tackled in the literature by integer linear programming approaches, by Greedy heuristics, and by different versions of a population-based iterated greedy algorithm. In this project, we first improve over the existing Greedy heuristics. This is done by implementing the rollout versions of these heuristics, and by testing them in a multistart framework in which they are applied in a probabilistic way. Second, we implement three versions of a biased random key genetic algorithm. The difference between these versions is found in the way in which individuals are decoded into feasible solutions to the problem. Moreover, we study the rollout versions of the corresponding decoders. Our results show that the developed algorithms can compete with the state of the art in the group of rather small-scale problem instances. However, with a growing size of the problem instances, our algorithms can not quite match the results of the current state-of-the-art algorithm. Nevertheless, our algorithms can potentially be improved in several different ways, which we explain in detail. Therefore, we believe that our algorithms should be further studied in future work.

Resum

El problema de la dominació independent ponderada és un problema NP-hard d'optimització combinatòria en grafs. Aquest problema només ha estat abordat a la literatura per enfocaments de programació lineal entera, heurístiques voraces i diferents versions d'algoritmes voraços iteratius basats en poblacions. En aquest projecte, primer apliquem una millora sobre les heurístiques voraces existents. Això ho fem implementant les versions *rollout* d'aquestes heurístiques i provant-les en un marc *multistart* on són aplicades de forma probabilística. En segon lloc, implementem tres versions d'un algorisme genètic esbiaixat de clau aleatòria. La diferència entre aquestes versions es troba en la forma en què els individus són descodificats en solucions viables del problema. Els resultats mostren que els algorismes desenvolupats poden competir amb els que són estat de l'art en el conjunt d'instàncies relativament petites. No obstant això, amb una mida creixent de les instàncies del problema, els nostres algorismes no poden arribar al nivell dels resultats obtinguts per l'algorisme més punter. Tot i això, els nostres algorismes poden ser millorats de moltes formes diferents, les quals expliquem en detall. Per tant, creiem que els nostres algorismes haurien de ser més estudiats en treballs futurs.

Resumen

El problema de la dominación independiente ponderada es un problema NP-hard de optimización combinatoria en grafos. Este problema sólo ha estado abordado en la literatura por enfoques de programación lineal entera, heurísticas voraces y diferentes versiones de algoritmos voraces iterativos basados en poblaciones. En este proyecto, primero aplicamos una mejora sobre las heurísticas voraces existentes. Esto lo hacemos implementando las versiones *rollout* de estas heurísticas y probándolas en un marco *multistart* donde son aplicadas de forma probabilística. En segundo lugar, implementamos tres versiones diferentes de un algoritmo genético sesgado de clave aleatoria. La diferencia entre estas versiones se encuentra en la forma en la que los individuos son decodificados en soluciones del problema. Los resultados muestran que los algoritmos desarrollados pueden competir con los que son estado del arte en el conjunto de instancias relativamente pequeñas. No obstante, con un tamaño creciente de las instancias del problema, nuestros algoritmos no pueden llegar al nivel de los resultados obtenidos por el algoritmo más puntero. Aún así, creemos que nuestros algoritmos pueden ser mejorados de muchas formas distintas, las cuales explicamos en detalle. Por lo tanto, creemos que nuestros algoritmos deberían ser más estudiados en trabajos futuros.

Acknowledgements

First of all, I would like to thank Dr. Maria J. Blesa for accepting to direct my Bachelor's Thesis and proposing the project herself. Without her guidance and support, this would not have been possible.

In this context, I would also like to thank Dr. Christian Blum for his involvement in this project as the co-director. He worked very hard to make my job easier.

I cannot but appreciate their commitment to this thesis.

Last but not least, I would like to thank my family and friends for their continuous and undeniable support through all these years.

Contents

List of Figures **iv**

List of Tables **v**

List of Algorithms **vi**

Acronyms **vii**

1 Introduction **1**

1.1 Terms and concepts 1
1.2 Problem formulation 2
1.3 State-of-the-art and related problems 4
1.4 Project description and outline 5
1.5 Scientific publications 6

2 Metaheuristics for Combinatorial Optimization **7**

2.1 Exact algorithms 7
2.2 Approximate algorithms 8
2.2.1 Heuristics 8
2.2.2 Metaheuristics 9
2.3 Hybrid algorithms 11

3 New Algorithmic Approaches for the WID Problem **13**

3.1 Greedy algorithms 13
3.1.1 Rollout mechanism 15
3.1.2 Multistart mechanism 16
3.2 Biased Random-Key Genetic Algorithm 19
3.2.1 Genetic Algorithms 19
3.2.2 Biased Random Key Genetic Algorithms 20
3.2.3 A BRKGA for the WID problem 21
3.2.4 Evaluation of an Individual 23
3.2.5 Rollout mechanism 26

4 Experimental evaluation **29**
4.1 Benchmark instances . . . 29
4.2 Tuning . . . 30
4.2.1 General considerations . . . 30
4.2.2 Tuning of the multistarts algorithms . . . 31
4.2.3 Tuning of BRKGAs . . . 31
4.2.4 Tuning of BRKGAs rollout version . . . 32
4.3 Results . . . 35

5 Final words **47**
5.1 Conclusions . . . 47
5.2 Future work . . . 47
5.2.1 Parallel computing . . . 48
5.2.2 Generating greedy individuals . . . 51
5.2.3 Other improvements . . . 52
5.3 Technical competences . . . 54

A Project management **57**
A.1 Project planning . . . 57
A.1.1 Schedule . . . 57
A.1.2 Methodology . . . 57
A.1.3 Task description . . . 58
A.1.4 Estimated Time . . . 60
A.1.5 Final planning . . . 60
A.2 Budget . . . 63
A.2.1 Human resources . . . 63
A.2.2 Hardware resources . . . 64
A.2.3 Software resources . . . 64
A.2.4 Direct costs . . . 64
A.2.5 Indirect costs . . . 65
A.2.6 Total budget . . . 65
A.2.7 Budget deviations . . . 65

B Sustainability report **67**
B.1 Environmental dimension . . . 67
B.2 Economic dimension . . . 67
B.3 Social dimension . . . 68
B.4 Sustainability matrix . . . 68

List of Figures

1.1 Examples of independent and dominating sets in undirected graphs 3
1.2 Example of the WID problem 4

3.1 Rollout procedure. 17
3.2 Terms of a Genetic Algorithm. 19
3.3 Transition from generation k to generation $k+1$ in a BRKGA. 22
3.4 Parametrized uniform crossover: mating in BRKGAs. 22
3.5 Decoding process of an individual (version 1). 24
3.6 Decoding process of an individual (version 2). 25

4.1 Boxplots for random graphs with a neutral weight scheme. 40
4.2 Boxplots for random graphs with a node-oriented weight scheme. 41
4.3 Boxplots for random graphs with an edge-oriented weight scheme. 42
4.4 Boxplots for random geometric graphs with a neutral weight scheme. . . . 43
4.5 Boxplots for random geometric graphs with a node-oriented weight scheme. 44
4.6 Boxplots for random geometric graphs with an edge-oriented weight scheme. 45

5.1 Mapping CUDA hardware model to software model. 51
5.2 Example: transformation from solution space to the hypercube (version 1). 53
5.3 Example: transformation from solution space to the hypercube (version 2). 54

A.1 Initial planning. Gantt chart. 61
A.2 Final planning. Gantt chart. 62

List of Tables

4.1 Results of tuning GREEDY1-MULTISTART 31
4.2 Results of tuning GREEDY2-MULTISTART 31
4.3 Results of tuning BRKGA-V1 32
4.4 Results of tuning BRKGA-V2 32
4.5 Results of tuning BRKGA-V3 32
4.6 Results of tuning BRKGA-V1-ROLLOUT 33
4.7 Results of tuning BRKGA-V2-ROLLOUT 33
4.8 Results of tuning BRKGA-V3-ROLLOUT 34
4.9 Numerical results for random graphs. Greedy algorithms. 36
4.10 Numerical results for random geometric graphs. Greedy algorithms. 37
4.11 Numerical results for random graphs. Genetic algorithms. 38
4.12 Numerical results for random geometric graphs. Genetic algorithms. 39

A.1 Summary of the estimated time spent on each task 63
A.2 Human resources budget. 63
A.3 Hardware resources budget. 64
A.4 Direct costs. 65
A.5 Indirect costs. 65
A.6 Total budget. 66

B.1 Sustainability matrix 68

List of Algorithms

1 Pseudo-code of Greedy1 and Greedy2 14
2 Pseudo-code of Greedy-Rollout 17
3 Pseudo-code of Greedy-Multistart 18
4 Pseudo-code of a GA 20
5 A BRKGA for the WID Problem 23
6 Pseudo-code of the rollout implementation of decoders 1 and 2. 27

Acronyms

ACO	Ant Colony Optimization. 11
BRKGA	Biased Random-Key Genetic Algorithm. 20, 21, 26, 30, 31, 33, 47, 50–52, 54
CMSA	Constuct, Merge, Solve & Adapt. 54
CO	Combinatorial Optimization. 7, 11, 13
CoEA	Co-Evolutionary Algorithm. 53
CP	Constraint Programming. 8
DP	Dynamic Programming. 8, 15
EA	Evolutionary Algorithms. 11, 19, 53
GA	Genetic Algorithm. 19, 20, 51
GECCO	Genetic and Evolutionary Computation Conference. 6
GRASP	Greedy Randomized Adaptive Search Procedures. 10
IG	Iterated Greedy. 10
ILP	Integer Linear Programming. 8
ILS	Iterated Local Search. 10
LS	Local Search. 9, 10
MILP	Mixed Integer Linear Programming. 8
RKGA	Random-Key Genetic Algorithm. 20, 21
SA	Simulated Annealing. 10
TS	Tabu Search. 10
WID	Weighted Independent Domination. 2–5, 9, 11, 13, 16, 19, 21, 47

Chapter 1

Introduction

Graphs have been, for years, a major topic of study in discrete mathematics and computer science. Since it was first introduced back in the 18th century, the subject of graph theory has made an incredible progress.

Nowadays, uncountable real-life problems can be modelled as graphs, and applications range from technical (computer science and mathematics) to social areas. However, many of these problems cannot be efficiently solved to optimality. This phenomenon accentuates with the arrival of the information era and Big Data, mostly due to the massive size of problem instances. Nonetheless, in practical uses, instead of spending computational resources looking for an optimal solution, a good enough (but potentially suboptimal) solution may often be found in a reasonable amount of time by using approximation methods.

In this project, we are going to tackle one of these hard optimization problems, called the "Weighted Independent Domination Problem".

In this first chapter, we introduce the reader to the subject under study. We start by explaining the basic concepts, allowing us to correctly define the tackled problem. Then, we present the current state of the problem and its applications, along with a review of the related literature. Finally, we define the main objective of this thesis.

1.1 Terms and concepts

Before describing the problem, let us introduce the necessary graph concepts:

Graph basics: An undirected graph $G = (V, E)$ is a set of vertices V and a set of undirected edges E. An edge $e = (u, v) \in E$ connects two vertices $u \neq v \in V$, and may be equivalently denoted by (v, u). It is said that $e = (u, v)$ is *incident* to u and v, and that these two vertices are the endpoints of e. Furthermore, the set of edges incident to a vertex v is denoted by $\delta(v)$. In addition, two edges are called *adjacent* if they share a common vertex. Similarly, two vertices are called adjacent if they share a common edge.

Neigborhoods: When referring to the neighborhood of a vertex v, we may refer to the *open neighborhood* $N(v) := \{u \in V \mid (v, u) \in E\}$, i.e. the vertices adjacent to v, or to the *closed neighborhood* $N[v] := N(v) \cup \{v\}$.

Given a vertex $u \in V$ and another $v \in V$ with $u \neq v$, u is called a neighbor of v if and only if $u \in N(v)$. As usual, the *degree* of a vertex is the number of edges incident to it, i.e. $deg(v) = |N(v)|$. Additionally, we define the *D-restricted neighborhood* of a vertex $v \in V \setminus D$ as the open neighborhood of v restricted to all its neighbors that are in D, i.e. $N_D(v) := N(v) \cap D$.

Dominating and independent sets: Given an undirected graph $G = (V, E)$, a *dominating set* is a subset $D \subseteq V$ such that each vertex v is either in D ($v \in D$) or has at least one neighbor in D ($\exists u \in N(v) \mid u \in D$). Furthermore, an *independent set* is a subset $I \subseteq V$ such that for any pair $v \neq u \in I$, it holds that they are not connected by an edge $e \in E$. An independent set is called *maximal* if by adding any extra vertex from $V \setminus I$ to I, the set would lose its property of being independent. Note that every maximal independent set is also a dominating set. Thus, a maximal independent set is also called an *independent dominating set*.

Examples for dominating and independent sets are shown in Figure 1.1. In (a) we can see an undirected graph, where circles represent vertices and the lines joining them represent edges. In (b), a possible independent set is constructed. A darker color indicates the vertices belonging to the set. In (c), the chosen vertices form a dominating set. Thicker lines indicate edges connecting vertices not in the dominating set to vertices in the set. Finally, (c) shows an independent and dominating set. Note that it is also maximal independent (by definition).

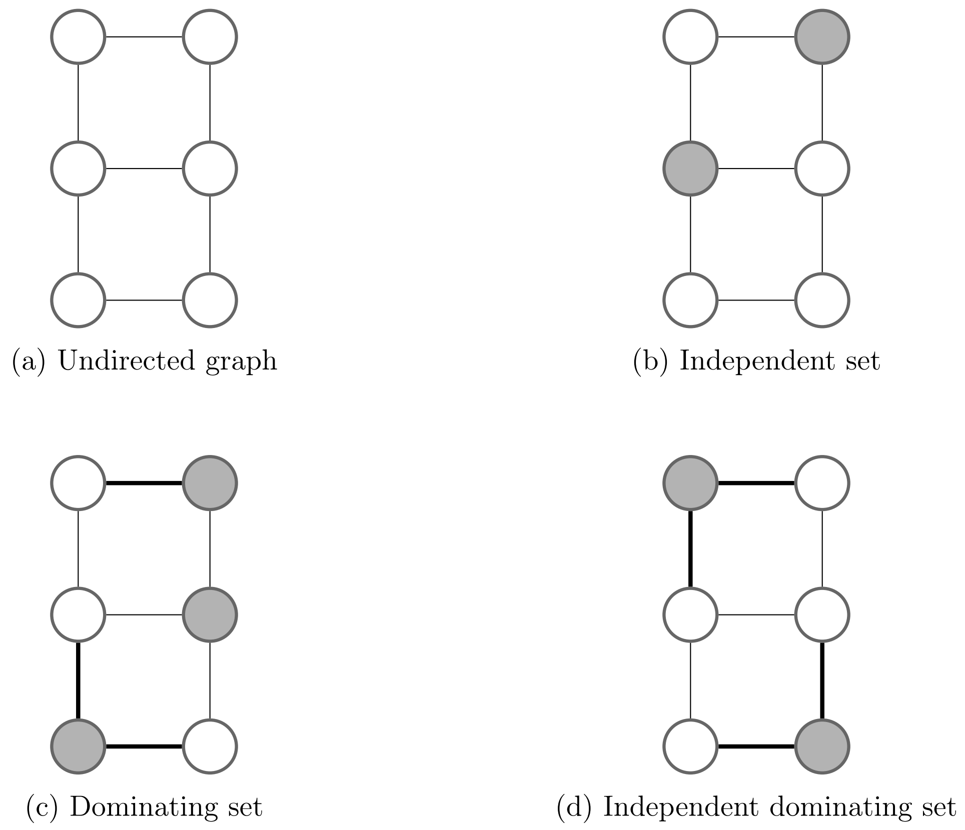
1.2 Problem formulation

The Weighted Independent Domination (WID) problem is an NP-hard combinatorial optimization[1] problem [13, 9] where we are given an undirected graph $G = (V, E)$ with vertex and edge weights (denoted by $w(v)$ and $w(e)$ respectively), which are integers greater or equal than 0. The WID problem consists in finding an independent dominating set D in G minimizing the following objective function:

$$f(D) := \sum_{v \in D} w(v) + \sum_{u \in V \setminus D} \min\{w(u, v) \mid v \in N_D(u)\} \tag{1.1}$$

The objective function value is calculated as the sum of the weights of the vertices in D plus the sum of the weights of the minimum-weight edges that connect the vertices not in D to the ones in D.

[1]Combinatorial optimization problems consist on finding an optimal object from a finite set. See Chapter 2 for further information.

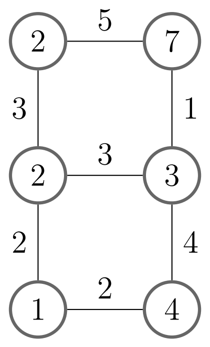

(a) Undirected graph (b) Independent set

(c) Dominating set (d) Independent dominating set

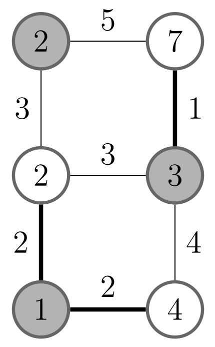
Figure 1.1: Examples of independent and dominating sets in undirected graphs

In Figure 1.2 we can see an Example. Consider the input graph in (a). Numbers inside nodes illustrate their weights. Similarly, edge weights are indicated by the value beside them. (b) shows the optimal solution to the WID problem. A darker colour is used to indicate the vertices that are part of the solution, and the minimum-weight edges connecting vertices not in the set to vertices from the set are marked with bold. In this case, the objective function value is 11, obtained by adding the weight of the selected vertices $(2 + 3 + 1)$ and the edge weights $(1 + 2 + 2)$.

Applications which can be modelled in terms of finding independent and/or dominating sets in graphs are abundant in real life settings. According to [30], finding minimum independent dominating sets has, in particular, applications in the context of clustering approaches and the placement of actors in their respective clusters in wireless networks. Another application is in the context of virtual backbone generation in mobile wireless ad-hoc networks. As pointed out in [23], the WID problem may arise in the context of clustering algorithms whenever a cost function is known for connecting non-selected nodes to cluster heads.



(a) Weighted undirected graph



(b) Optimal solution to the WID problem

Figure 1.2: Example of the WID problem

1.3 State-of-the-art and related problems

Hard combinatorial optimization problems in which solutions are subsets of the nodes of a given input graph are abundant in the scientific literature.

The WID problem was initially introduced by Chang in [9], where he records an unpublished result of 20 years ago stating that the problem is NP-complete for chordal graphs. Later on, Chang, Liu and Wang proposed a linear time algorithm for the problem in series-parallel graphs [27].

The most recent paper in the WID at the time of the inscription of this bachelor thesis project was authored by its co-director Christian Blum [23], being one of the fewest tackling the problem from a non-theoretical point of view. The contribution consisted of the development of three integer linear programming models, two greedy heuristics and a population-based iterated greedy metaheuristic applied in two different ways, with the respective experimental evaluation.

Although not much work has been done on this specific problem, others closely related need to be considered in this thesis:

- Maximum independent sets. The currently most successful algorithms tackling this problem include the general swap-based multiple neighbourhood tabu search proposed in [17], the fast local search routines in [1] and the iterated local search presented in [22], currently the best metaheuristic for the node-weighted variant of the problem.

- Minimum dominating sets. Some of the currently best metaheuristics solving this problem are an ant colony optimization and a genetic algorithm from [25], a hybrid approach combining iterated greedy algorithms and an integer linear programming solver in a sequential way [8], a local search based on two-level configuration checking from [28] or a hybrid evolutionary algorithm in [10].

- Minimum independent dominating sets. Recently tackled by an adaptive search procedure in [30] and a memetic algorithm from [29].

1.4 Project description and outline

Objective

Our main goal is to develop new heuristics that improve the performances of the ones already existing, as well as modifying them by applying new techniques. Our work will be centred on the implementation of a genetic algorithm, a metaheuristic inspired by the process of natural selection.

In summary, we hope to find some algorithmic technique which achieves a new state-of-the-art for the WID problem. Although improvement will be sought, it is not a mandatory requirement for this bachelor thesis.

Scope

As this project is a research thesis, no final product will be presented. Like any other investigation work, the scope will entirely depend on the performance of our solutions.

Good quality algorithms will allow us to go further into the problem while testing potential improvements to be applied. On the other hand, poor results will lead to a change of perspective and a look for other potential solutions. Although this may seem a hindrance to success, the wide range of possibilities will let us explore different paths if we get to a dead end.

Project outline

The document is structured in the following way:

1. In Chapter 2, we explain what a Combinatorial Optimization problems are and introduce different methods to solve them.

2. In Chapter 3, an extensive description of our developed algorithms is presented.

3. In Chapter 4, we present and evaluate the obtained results.

4. In Chapter 5, we conclude the thesis by making an overview of the work and introducing some ideas for future work,

5. Appendices A and B contain the planning of the project and its sustainability analysis, respectively.

1.5 Scientific publications

A paper about one of our developed approaches (specifically the one proposed at Section 3.2.4) has been accepted and will be presented at the Student Workshop of the Genetic and Evolutionary Computation Conference (GECCO) 2019 in Prague. GECCO is one of the best conference from the field of evolutionary computation. It is included in the CORE conference ranking database and ranked as an A-conference (where the range, with increasing quality, goes from C to A*).

Chapter 2

Metaheuristics for Combinatorial Optimization

As explained in [7], a Combinatorial Optimization (CO) problem $\mathcal{P} = (\mathcal{S}, f)$ consists of finding an optimal object s^* within a finite set of objects $\mathcal{S}$. An objective function $f : \mathcal{S} \mapsto R$ assigns a value to each of the objects $s \in \mathcal{S}$. The goal is, then, to find an object s^* with a minimal (or maximal) cost value. Furthermore, the set of all possible components of which solutions to $\mathcal{P}$ are composed is denoted by $\mathcal{C}$. Thus, a valid solution S to the problem is a subset of the complete set of solution components $\mathcal{C}$ ($S \subseteq \mathcal{C}$).

Problems of this type are abundant in scientific literature. Well-known application areas where CO problems can be found are routing, scheduling, packing, network design and bioinformatics.

Unfortunately, most of these CO problems are in NP, thus solving them to optimality is very difficult in practice. Over the last decades, a wide variety of algorithms and mathematical approaches have emerged to tackle CO problems. These can be essentially classified in two types: *exact* or *approximate* methods.

2.1 Exact algorithms

These type of algorithms guarantee that the optimal solution to a problem is found in bounded time. However, unless $P = NP$, an exact algorithm trying to solve an NP-hard CO problem cannot run faster than worst-case exponential time. This issue leads to remarkably high computation times for practical purposes, specially as the size of the instances grows.

Nevertheless, the property that a given CO problem is NP-hard does not imply that an exact algorithm cannot be applied in practice. This method might obtain proven optimal solutions for instances with a small size or exploitable structures in reasonable time.

Some of the most popular exact approaches are:

- Tree Search Methods.

Consist on exhaustively enumerating all potential solutions by recursively dividing the search space into disjunct subspaces. This partitioning can then be associated with a rooted directed tree, where different graph traversal techniques can be applied on the search for the optimal solution.

- Dynamic Programming (DP).

  Follows the same partitioning principle as Tree Search Methods, but instead makes use of the overlapping properties of some subproblems by storing their results in order to probably reuse them lately, avoiding repeated calculations.

- Integer Linear Programming (ILP).

  Seeks to minimize a linear cost function subject to a set of linear equalities and inequalities in which variables are restricted to be integers.Mixed Integer Linear Programming (MILP) models also allow continuous variables.

- Constraint Programming (CP).

  Technique oriented to solving problems involving complicated constraints.

2.2 Approximate algorithms

Approximation algorithms, in contrast to the exact approaches, sacrifice the guarantee of finding optimal solutions for the sake of obtaining relatively close solutions in a feasible amount of time.

The interest in these methods has increased in recent decades. Although they do not assure optimal solutions, it does not mean they cannot be obtained. Furthermore, they can achieve high-quality solutions in instances which would be intractable with exact approaches.

These can be essentially divided into two types: *heuristics* and *metaheuristics*.

2.2.1 Heuristics

Heuristics are fast computational methods that can obtain acceptable solutions taking full advantage of the particularities of the tackled problem. Therefore, they are problem-dependent techniques. However, heuristic approaches are usually too greedy, thus getting easily trapped in local minima. Heuristic methods can be classified in two big families:

- Constructive heuristics:

  In constructive heuristics, solutions are build from scratch by adding, at each iteration, a solution component to a partial solution, which is initially empty. This procedure is repeated until a stopping criteria is met, usually when the actual solution cannot be further extended.

Greedy algorithms are particular cases of constructive heuristics. They follow the problem solving heuristic of making the locally optimal choice at each stage with the intent of finding a global optimum. In many problems, a greedy strategy does not usually produce an optimal solution, but nonetheless a greedy heuristic may yield locally optimal solutions that approximate a globally optimal solution in a reasonable amount of time. In general, greedy algorithms have five components:

- a candidate set, from which a solution is created,
- a selection function, which chooses the best candidate to be added,
- a feasibility function to determine if a candidate can be used to contribute to a solution,
- an objective function, which assigns value to a solution (partial o complete) and
- a solution function, which will indicate when the solution is complete.

Greedy algorithms produce good solutions on some mathematical problems, but not on others. Most problems for which they work will have two properties: the so-called *Greedy choice property*[1] and *Optimal substructure*.[2]

Some of the algorithms proposed for the WID problem in this work are based on greedy strategies or make use of them.

- Local Search (LS):

  These algorithms, on the contrary, start from an initial solution and replace it, at each step, with a better one from its neighborhood. This process is repeated until no better solution exists in the neighborhood.

2.2.2 Metaheuristics

Metaheuristics, on the other hand, are problem-independent techniques tan can be applied to a large range of problems. As their name suggests, they are superior-lever frameworks that try to use heuristics or other methods to effectively explore the search space and thus hopefully getting an optimal or near-optimal solution. In contrast to standard heuristics, they use a certain trade-off between randomization and local search, which allows them to explore more thoroughly the solution space. Although they are problem-independent methods, note that it is nonetheless necessary to adapt the technique to the problem at hand.

[1]The choice made by a greedy algorithm may depend on choices made so far, but not on future choices or all the solutions to the subproblem. It iteratively makes one greedy choice after another, reducing each given problem into a smaller one. In other words, a greedy algorithm never reconsiders its choices. This is the main difference from dynamic programming, which is exhaustive and is guaranteed to find the solution. After every stage, dynamic programming makes decisions based on all the decisions made in the previous stage, and may reconsider the previous stage's algorithmic path to solution.

[2]A problem exhibits optimal substructure if an optimal solution to the problem contains optimal solutions to the sub-problems.

Based on Local Search

Simple LS methods are usually characterized by getting easily stuck in local minima. Furthermore, the quality of the obtained solution is strongly subjected to the starting point of the search process and, generally, the basin of attraction of the global minimum is not known.

Therefore, different metaheuristic techniques have been developed in order to include an *exploration* component to the iterative improvement, whose purpose is to guide the exploration through the search space trying to achieve better and better local minima. For these reason, though, some stopping criteria other than finding a local minumum is needed. Reaching a maximum number of iterations or finding a solution of sufficient quality are some of the most used termination criteria.

Some of the most important local-search based metaheuristics are the following ones. Since these techniques are not directly related with the algorithms proposed in this work, only the main idea of their behaviour is cited. For a more detailed description of each of them, the reader is referred to the bibliographic citations:

- Greedy Randomized Adaptive Search Procedures (GRASP) [12]:

  Repeatedly generates stochastic greedy solutions, subsequently improving them through the application of a local search method.

- Iterated Greedy (IG) [26]: Iterates through the partial destruction of a complete solution in a probabilistic way and its subsequent reconstruction, making use of the same construction mechanism as for the generation of the initial solution.

- Iterated Local Search (ILS) [21]:

  LS method that, at each iteration, starts with a solution obtained by randomly perturbing a previous one. This alteration should move the search to another basin of attraction while staying close to the neighborhood of the initial solution. The obtained solution is then improved by using a standard LS method.

- Simulated Annealing (SA) [19]:

  Starting from an initial random solution, at each step, the algorithm randomly selects another solution from the neighborhood. If the new solution is better than the current one, the algorithm moves to that point in the search space. Otherwise, the new solution is accepted with a given probability that decreases as the algorithm explores the search space.

- Tabu Search (TS) [14]:

  Enhances the performance of a standard local search by allowing moves to worse positions in the search space. Furthermore, some prohibitions in the neighborhood are introduced based in previously visited solutions, making use of a short-term memory (also called a *tabu list*).

Population-based

These metaheuristics deal with multiple solutions rather than with a single one. This set of solutions is then modified by the application of certain operators to produce subsequent populations. Population-based metaheuristics are usually inspired by natural methods for effectively exploring the search space.

The two most studied population-based metaheuristics are:

- Ant Colony Optimization (ACO) [11]:

  A colony of agents (artificial ants) construct solutions by moving on the search space guided by pheromone values that are updated at runtime.

- Evolutionary Algorithms (EA) [2]:

  Inspired by the evolution of living beings and their environmental adaptation, an initial population of individuals is generated and altered through several iterations by the recombination and mutation of their genetic information.

Some of the algorithms proposed for the WID problem in this work are based on evolutionary algorithms; more precisely, in a relatively new type of Genetic Algorithms called Biased Random-Key Genetic Algorithms. The reader is referred to the next chapter for a detailed description of them.

2.3 Hybrid algorithms

A hybrid algorithm is one that combines two or more algorithms for solving the same problem. The goal is to take profit of the desired features of each, making the overall algorithm perform better than its individual components. These algorithms are widely used in optimization problems, specially making use of recursive algorithms (such as *divide and conquer*) to slice the problem into smaller instances, where a better performing (but time-consuming) one is used. The main motivation behind these approaches, then, is exploiting the complementary character of different optimization techniques (or *synergy*).

One example could be the hybridization of metaheuristics with other techniques, in particular with the above sketched exact methods. This particular case has proven effective in many CO problems [6]. Unfortunately, developing hybrid approaches is a difficult task, requiring expertise in different areas. In addition, hybrid algorithms seem to be problem-dependent, which means that a given algorithm might perform greatly for specific problems but poorly on others. They are, in general, nontrivial to generalize.

For further information about optimization and hybrid metaheuristics, we refer the interested reader to [7].

Chapter 3

New Algorithmic Approaches for the WID Problem

As previously mentioned, the WID problem is a CO problem. Given an undirected graph $G = (V, E)$, $\mathcal{S}$ corresponds to all possible dominating and independent sets of vertices in G. The objective function value of a set $S \in \mathcal{S}$ is calculated by Equation 1.1. Moreover, the set $\mathcal{C}$ is equivalent to the set of vertices from the input graph. In addition, the vertices forming an independent dominating set D compose the set of solution components contained in S.

In this chapter, we present an extensive description of the new algorithms that we have designed for solving this problem, making use of approximate methods: starting with constructive heuristics and finishing with some variants of an evolutionary algorithm.

3.1 Greedy algorithms

Greedy algorithms (also called Greedy heuristics) are probably the most well-known example of constructive heuristics, which are, usually, the fastest approximate methods.

Greedy heuristics make use of the so-called greedy function to determine which component is added at each step. This function measures the estimated quality of a solution component, and serves as an indicator for the effect on the objective value if the component would be included into the solution. Before describing our developed approaches, let us introduce two greedy heuristics from [23] that will be used in the forthcoming approaches. Both algorithms (henceforth denoted as GREEDY1 and GREEDY2) share the same framework, but they differ in the heuristic function employed to determine which vertex is chosen at each step.

Given an input graph $G = (V, E)$, the algorithm starts with an empty solution $S = \emptyset$ and the so-called *remaining graph* $G' = (V', E')$, which is initially a copy of G.

At each iteration, exactly one vertex v from the remaining graph G' is added to S. After that, all the vertices from $N_{G'}[v]$ (the closed neighborhood of v in G') are removed from V'. Additionally, all their incident edges are removed from E'. By doing so, we ensure that S maintains the property of being independent.

The algorithm finishes when V' is empty. At this point, we can clearly see that S is independent and dominating, as the vertices that were not added to the solution were those adjacent to the chosen ones.

The pseudo-code of this procedure is shown in Algorithm 1.

Algorithm 1 Pseudo-code of GREEDY1 and GREEDY2

```
1: input: an undirected weighted graph $G = (V, E, w)$
2: $S := \emptyset$
3: $G' := G$
4: while $V' \neq \emptyset$ do
5:     Choose $v^* \in V'$ by using greedy function $\eta_1$ (for GREEDY1) or $\eta_2$ (for GREEDY2)
6:     $S := S \cup \{v^*\}$
7:     Remove from $G'$ all nodes from $N_{G'}[v^*]$ and their incident edges
8: end while
9: output: An independent dominating set $S$ of $G$
```

Greedy function η_1 of GREEDY1. In this heuristic, vertices with a high degree in the remaining graph G' and a low vertex weight are preferred. The function is defined as follows:

$$\eta_1(v) := \frac{deg_{G'}(v)}{w(v)}$$

At each step, the vertex $v \in V'$ with the highest η_1-value is chosen, so $v^* := \text{argmax}\{\eta_1(v) \mid v \in V'\}$ (see LINE 5 of Algorithm 1).

Note that this heuristic does not consider edge weights. They are only taken into account when calculating the objective function of the final solution S.

Greedy function η_2 of GREEDY2. In contrast to GREEDY1, this second heuristic is designed to consider the edge weights while constructing the solution. Before describing the function, it is necessary to introduce the following notations.

First, let $w_{\max}$ be the maximum weight of all the edges, hence $w_{\max} := \max\{w(e) \mid e \in E\}$. Second, let S be a partial solution. Then, an *auxiliary objective function* $f^{\text{aux}}(S)$ is defined as $\sum_{v \in V} c_S(v)$, where $c_S(v)$ is called the *contribution* of vertex v with respect to the partial solution S and is defined as follows:

$$c_S(v) := \begin{cases} w(v) & \text{if } v \in S \\ w_{\max} & \text{if } v \notin S \text{ and } N(v) \cap S = \emptyset \\ \min\{w(e) \mid e = (v,u), u \in S\} & \text{if } v \notin S \text{ and } N(v) \cap S \neq \emptyset \end{cases}$$

Then, greedy function η_2 is defined as follows:

$$\eta_2(v) := f^{\text{aux}}(S \cup \{v\})$$

At each step, the vertex $v \in V'$ with the lowest η_2-value is chosen, so $v^* := \text{argmin}\{\eta_2(v) \mid v \in V'\}$ (see LINE 5 of Algorithm 1). Note that, in the case of S being a complete solution, it holds that $f(S) = f^{\text{aux}}(S)$. For more information, we refer the interested reader to [23].

In the following subsections, two modifications of these greedy algorithms are detailed. Henceforth, when using η, we may refer to η_1 or η_2, depending on the greedy heuristic used (GREEDY1 or GREEDY2 respectively).

3.1.1 Rollout mechanism

Rollout is a form of sequential optimization in DP (see Section 2.1 for further information). Using some given heuristic, the rollout mechanism makes use of this heuristic in a subordinate way, often obtaining much better performance than the original heuristic. It uses a lookahead approach and benefits of the repeated application of the greedy heuristic in an intelligent way [4].

The rollout mechansim is explained in Bertsekas (2013) as follows. Consider the minimization of a function $f(x_1, ..., x_n)$ of discrete variables $x_1, ..., x_n$ that take values each from some finite set. This minimization task can be rephrased as a sequence of minimization tasks as follows. First, the optimal value of x_1 (denoted by x_1^*) is obtained by minimizing $F_1(x_1) = \min\limits_{x_2,...,x_n} f(x_1, ..., x_n)$. Afterwards, x_1^* is fixed and the same is done for x_2, resulting in x_2^*, etc. In general, for every $k = 1, ..., n$, given the optimal values $x_1^*, ..., x_{k-1}^*$, the optimal value of x_k can be found by the minimization

$$x_k^* = \underset{x_k}{\text{argmin}}\ F_k(x_1^*, ..., x_{k-1}^*, x_k) \tag{3.1}$$

where

$$F_k(x_1^*, ..., x_{k-1}^*, x_k) = \min_{x_{k+1},...,x_n} f(x_1, .., x_n) \tag{3.2}$$

In DP, functions F_k are called *optimal cost-to-go* functions. However, calculating and storing them in tables becomes intractable in practice, especially in the context of large instances.

In the rollout approach, the F_k functions are *approximated* by other, more affordable, alternatives. In particular, function F_k, for $k = 1, ...n$, is exchanged by a much cheaper, so-called, *base heuristic* H_k that produces a (possibly suboptimal) value $\hat{x}_k$. Hence, the rollout algorithm achieves a (possibly suboptimal) value $\hat{x}_k$ by replacing F_k with H_k in Eq. (3.1):

$$\hat{x}_k = \underset{x_k}{\text{argmin}}\ H_k(\hat{x}_1, ..., \hat{x}_{k-1}, x_k) \tag{3.3}$$

Notice that the result obtained in this way is at least as good as constructing the solution by just applying the base heuristic once to the whole problem. Although this procedure is done in expense of a considerable increase in computational time and space, results show that a significant improvement is obtained over the application of just the base heuristic.

The rollout algorithm will now be formalized for the WID problem.

Given an undirected graph $G = (V, E)$, let $S_k \subset V$ be the partial solution at iteration k (which are the set of vertices chosen at previous iterations) and V' be the set of vertices from the remaining graph G'.

Furthermore, $\mathcal{H}$ denotes a heuristic construction algorithm which, given a partial solution S_k, constructs a complete solution (a dominating and independent set in G). Henceforth, $\mathcal{H}$ will also be referred to as the base heuristic. In our case, the heuristics used are Greedy1 and Greedy2.

Finally, let $H(S)$ be the final solution obtained by applying the base heuristic $\mathcal{H}$ to the corresponding partial solution S. Recall that $f(H(S))$ denotes its objective function value. Given a vertex v, we denominate the *projection of v under $\mathcal{H}$ in iteration k*, denoted by $p_k(v)$, to the objective function value of the final solution obtained by applying the base heuristic to the partial solution $S_k \cup \{v\}$. Therefore, $p_k(v) := f(H(S_k \cup \{v\}))$. In words, the projection is the objective value of the solution obtained if we would add the vertex to the current partial solution and completed it using one of the greedy heuristics.

Our algorithm, then, starts with an empty solution $S := \emptyset$ and, at each iteration k, adds the vertex in the remaining set V' with the lower projection value p_k. The vertices from the closed neighborhood of v^* in G' (and their incident edges) are then removed from G'. We proceed until the set of remaining vertices V' is empty. The pseudo-code of this algorithm can be found in Algorithm 2. Figure 3.1 visually describes the process carried out during an iteration.

Note that this algorithm only differs from the base greedy heuristics in the way it selects which vertex is added at each iteration, adding a superior layer to the procedure.

In the case of Greedy1, the algorithm stores a partial objective function value during the construction of the solution. At each iteration, the algorithm adds the weight of the selected vertices. When the solution is completed, the objective function value is increased by the weight of the minimum-cost edges that connect vertices not in the solution set to vertices in the set. Then, as the value starts at 0 and increases at each iteration, we can stop the calculation of a given projection if its partial objective value is higher than the best-so-far projection. This phenomenon is called a *cut*. Note that this cannot be applied to Greedy2, as it makes use of an auxiliary objective function that starts at a maximum theoretical value and decreases as vertices are added to the solution.

Two rollout versions of the two Greedy base-heuristics are henceforth called Greedy1-Rollout and Greedy2-Rollout.

3.1.2 Multistart mechanism

In this approach, we make use of the two previously mentioned greedy heuristics in order to generate probabilistic solutions to the problem. The main idea is to constantly build solutions from scratch and store the best one encountered, which will be returned by the algorithm.

For this purpose, we require values for two parameters: the determinism rate $d_{\text{rate}} \in [0, 1]$ and the list size $l_{\text{size}} \in \mathbb{N}^+$, which determine the degree of greediness of the constructed

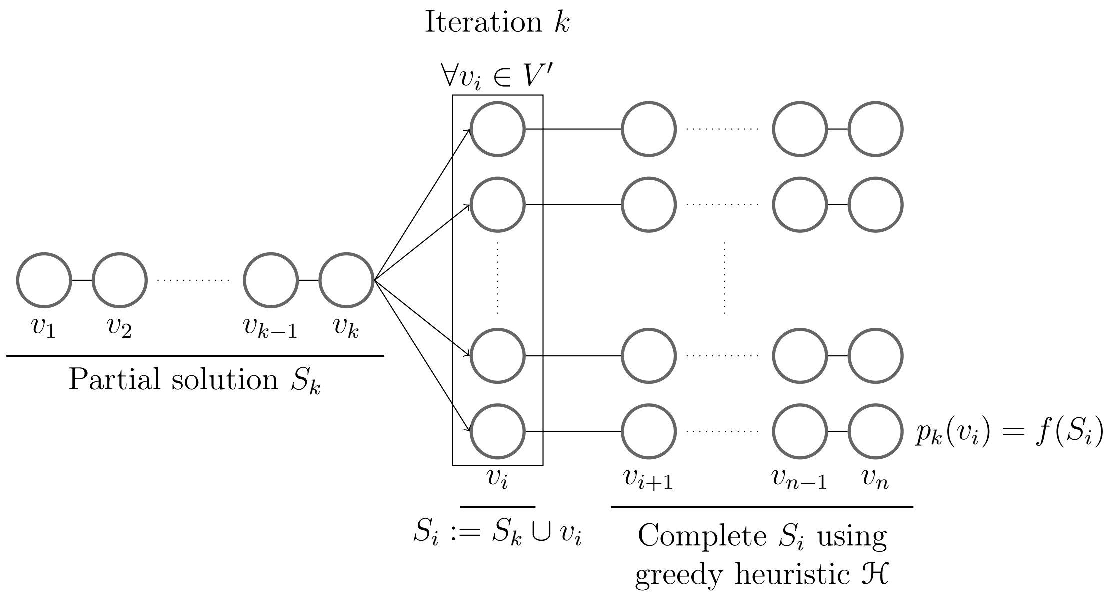


Figure 3.1: Rollout procedure.

Algorithm 2 Pseudo-code of GREEDY-ROLLOUT

```
1: input: an undirected weighted graph $G = (V, E, w)$
2: $S := \emptyset$
3: $G' := G$
4: while $V' \neq \emptyset$ do
5:     $v^* = \text{argmin}\{p_k(v) \mid v \in V'\}$        {NOTE: Iteration k}
6:     $S := S \cup \{v^*\}$
7:     Remove from $G'$ all nodes from $N_{G'}[v^*]$ and their incident edges
8: end while
9: output: An independent dominating set $S$ of $G$
```

solutions.

The algorithm starts with an empty set $S_{\text{best}} := \emptyset$, representing the best solution found so far. Initially, as S_{best} is not a valid solution to the problem, we suppose $f(S_{\text{best}})$ returns the maximum possible objective function value.

The procedure of building a solution differs from the standard greedy algorithms in the way it selects the vertex added to the solution at each construction step (see LINE 5 of Algorithm 1).

We also start with an empty set $S := \emptyset$. At each iteration, a vertex from the remaining graph is added to the solution, which is chosen in the following way. First, a uniformly distributed random number $r \in [0, 1]$ is generated. If $r \leq d_{\text{rate}}$, we add the vertex with the best greedy function value to the set S. Otherwise, we add one at random from the $\min\{l_{\text{size}}, |V'|\}$ vertices in V' with the best greedy function value, which are the ones with the maximum η_1 value in case of GREEDY1 or the minimum η_2 value in case of GREEDY2. Note that using a d_{rate} value of 1 is equivalent to constructing the solution using the standard greedy heuristics. Once finished, the vertices from $N_{G'}[v]$ are removed from the remaining graph. This procedure is repeated until we run out of vertices in G'.

If the obtained solution is better than the best-so-far ($f(S) < f(S_{\text{best}})$), then the newly build solution becomes the best-so-far ($S_{\text{best}} := S$).

We keep building probabilistic solutions until a certain time limit is reached, and S_{best} is then returned. The pseudo-code of this algorithm can be found in Algorithm 3 .

Algorithm 3 Pseudo-code of GREEDY-MULTISTART

```
 1: input: an undirected weighted graph $G = (V, E, w)$
 2: $S_{\text{best}} := \emptyset$        {NOTE: Suppose $f(S_{\text{best}})$ is, initially, the maximum possible value}
 3: while computation time limit not reached do
 4:    $S := \emptyset$
 5:    $G' := G$
 6:    while $V' \neq \emptyset$ do
 7:       Random $r \sim U(0, 1)$
 8:       if $r \leq d_{\text{rate}}$ then
 9:          Choose $v^* \in V'$ by using greedy function $\eta$
10:       else
11:          Choose $v^* \in V'$ at random within the $l_{\text{size}}$ vertices with the best greedy function
             values
12:       end if
13:       $S := S \cup \{v^*\}$
14:       Remove from $G'$ all nodes from $N_{G'}[v^*]$ and their incident edges
15:    end while
16:    if $f(S) < f(S_{\text{best}})$ then
17:       $S_{\text{best}} := S$
18:    end if
19: end while
20: output: an independent dominating set $S_{\text{best}}$ of $G$
```

Two multistart versions were constructed (one for each greedy heuristic), henceforth called GREEDY1-MULTISTART and GREEDY2-MULTISTART.

3.2 Biased Random-Key Genetic Algorithm

In the forthcoming subsections, a general explanation about Genetic Algorithms is provided, along with a more specific description of the sub-type used in this thesis. Then, we present how this method was applied to the WID problem and discuss different approaches.

3.2.1 Genetic Algorithms

A Genetic Algorithm (GA) is a metaheuristic inspired by the process of natural selection (survival of the fittest) introduced by John Holland in [16]. It is frequently used to generate high-quality (optimal or near optimal) solutions to CO problems. It belongs to the larger class of EA (see Section 2.2.2 for further information).

In a GA, we have a set of *individuals* called *population*. These individuals are potential solutions to the problem. Each individual is encoded as a vector or string of variables (*genes*), called *chromosome*. Each gene can take on a value, called *allele*. See Figure 3.2 for a visual representation. Furthermore, individuals assigned a *fitness* value, which measures how suitable they are. Commonly, the value of the objective function in the optimization problem being solved is used.

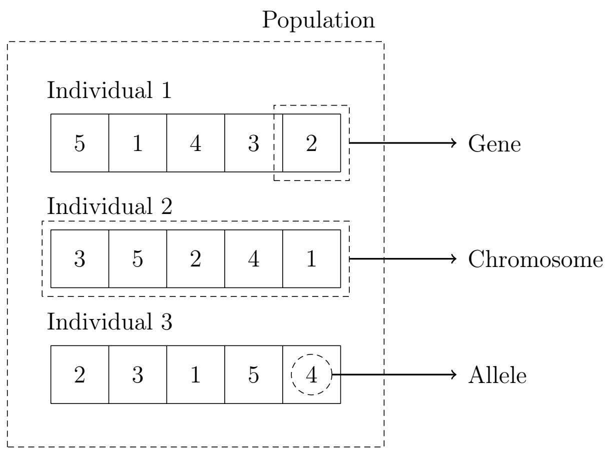


Figure 3.2: Terms of a Genetic Algorithm.

The algorithm starts with an initial population (usually randomly generated individuals) and then evolves it over a number of *generations*. At each generation, a new population is created by producing *offspring* of the current population, with the fittest individuals (the ones with highest fitness) having more probability to reproduce and pass on their characteristics to the next generation. Thus, each successive generation is more suited than the previous ones. The algorithm terminates when some defined criteria is met (frequently after a fixed number of generations or a satisfactory level of quality is reached).

A GA uses three main operators to evolve the population:

Algorithm 4 Pseudo-code of a GA

```
1: Generate initial population
2: Compute fitness
3: while termination condition not satisfied do
4:     Select parents from population
5:     Generate offspring population by crossover
6:     Apply mutation to the generated individuals
7:     Compute fitness of the new population
8: end while
```

- **Selection**: In each generation, a portion of the population is selected to pass their genes to successive generations by producing offspring. These individuals, called *parents*, are picked depending on their fitness value (fitter solutions are more likely to be chosen). Typical selection methods are *roulette wheel* or *tournament* selection.
- **Crossover**: Once the parents are chosen, pairs are combined to form children for the next generation. It is analogous to what happens during sexual reproduction in biology. The genetic information of the two parents (genes) are exchanged, creating a new individual (offspring). Typical crossover methods are single-point, two-point or uniform crossover.
- **Mutation**: In order to avoid premature convergence and maintain diversity within the population, random changes are applied to the offspring individuals. Usually, some genes in a chromosome are altered according to a defined probability.

A pseudo-code for the GA can be found in Algorithm 4

3.2.2 Biased Random Key Genetic Algorithms

A Biased Random-Key Genetic Algorithm (BRKGA) [15] is a steady-state genetic algorithm and a variant of a Random-Key Genetic Algorithm (RKGA) [3]. A RKGA differs from a classic GA in the following:

- Individuals are represented as a vector of real numbers between 0 and 1. A deterministic algorithm, the so-called *decoder*, is used to translate any chromosome to a valid solution of the tackled problem, where the objective value (or fitness) of the individual can be computed. Note, then, that this type of genetic algorithm consists of a problem-independent part (the evolution of the population) and a problem-dependent part (the decoder). This allows for the reuse of code, as only the problem-dependent part has to be modified when changing between problems.
- It uses an elitist strategy. The population is divided in two groups: *elite* individuals (those with the highest fitness values) and non-elite individuals. All of the elite individuals at each generation are preserved for the subsequent. The purpose is to

store good quality solutions found during previous iterations to positively influence future generations.

- Mutation is implemented by introducing *mutants* (random generated individuals) into the population at each generation. They are usually generated in the same way as in the initial population, thus can be decoded into valid solutions of the problem.

A BRKGA differs from a RKGA in the way parents are selected for mating. In a RKGA, both are selected at random from the entire population. However, in BRKGAs, one parent is always an elite individual.

3.2.3 A BRKGA for the WID problem

We will now describe more precisely the problem-independent part of the BRKGA applied to the WID problem, whose pseudo-code is provided in Algorithm 5. Our BRKGA depeds on several parameters:

- The number of genes in the chromosome of an individual, denoted by n.
- The number of individuals in the population, denoted by p_{size}.
- The fraction of elite individuals in the population, denoted by p_e.
- The fraction of mutants introduced at each generation, denoted by p_m.
- The probability that a given offspring inherits the allele from its elite parent, denoted by $prob_{\text{elite}}$. This value is required to be, at least, 0.5. We will see its purpose later.

The algorithm starts by generating a population P (made up of p_{size} random generated individuals) in $\mathsf{GenerateInitialPopulation}(p_{\text{size}})$. Each individual $\pi \in P$ is a vector of length $n = |V|$ (where V is the set of vertices from the input graph). Allele i of π (denoted by $\pi(i)$) is generated uniformly at random in the real interval $[0, 1]$.

Afterward, the fitness of each individual of the initial population is evaluated in function $\mathsf{Evaluate}(P)$. Each one is assigned an objective function value (denoted by $f(\pi)$) by the decoder, which is the problem-dependent part. Different decoder implementations are discussed in section 3.2.4.

At each iteration, the algorithm performs the following actions, which can be visualised in Figure 3.3:

1. First of all, the $\max\{\lfloor p_e \cdot p_{\text{size}} \rfloor, 1\}$ individuals with the best fitness values are copied to P_e (elite population) in function $\mathsf{EliteSolutions}(P, p_e)$.
2. Second, a set of $\max\{\lfloor p_m \cdot p_{\text{size}} \rfloor, 1\}$ so-called mutants are generated and stored in P_m (mutant population), produced in the same way as the individuals from the initial population.

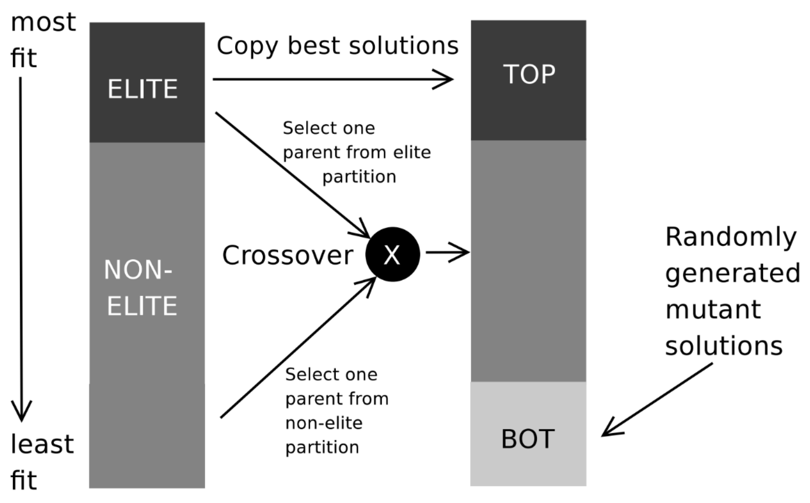

most fit
least fit
ELITE
NON-ELITE
Copy best solutions
Select one parent from elite partition
Crossover
X
Select one parent from non-elite partition
TOP
BOT
Randomly generated mutant solutions

Figure 3.3: Transition from generation k to generation $k+1$ in a BRKGA. Extracted from [15].

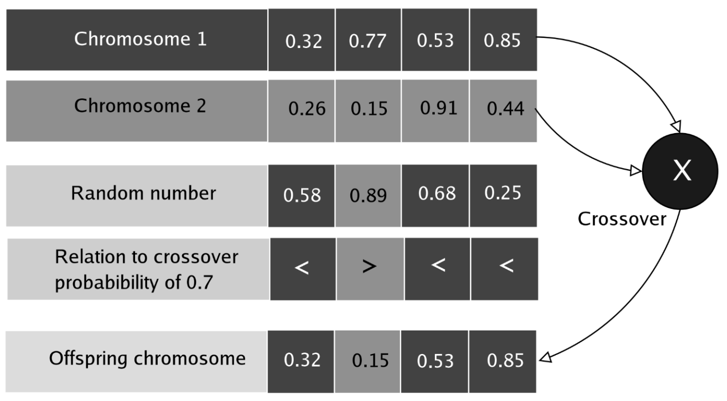
3. Then, $p_{\text{size}} - |P_e| - |P_m|$ offspring individuals are generated by crossover in function $\mathsf{Crossover}(P, P_e, prob_{\text{elite}})$ and stored in P_c (crossover population). A *parametrized uniform crossover* is used, proceeding as follows:

   (a) An elite parent π_{e} is chosen uniformly at random from P_e.

   (b) A non-elite parent π_{ne} is chosen uniformly at random from $P \setminus P_e$

   (c) An offspring π_{off} is generated by combining π_{e} and π_{ne}. The value of $\pi_{\text{off}}(i)$ is set to $\pi_{\text{e}}(i)$ with probability $prob_{\text{elite}}$, and to $\pi_{\text{ne}}(i)$ otherwise. Therefore, the offspring individual is more likely to inherit characteristics from its elite parent.

Figure 3.4 shows an example of this procedure. Note that, since p_e is required to



Chromosome 1	0.32	0.77	0.53	0.85
Chromosome 2	0.26	0.15	0.91	0.44
Random number	0.58	0.89	0.68	0.25
Relation to crossover probabibility of 0.7	<	>	<	<
Offspring chromosome	0.32	0.15	0.53	0.85

Figure 3.4: Parametrized uniform crossover: mating in BRKGAs. Extracted from [15].

Algorithm 5 A BRKGA for the WID Problem

```
1: input: an undirected weighted graph $G = (V, E, w)$
2: input: parameter values for $p_{\text{size}}$, $p_e$, $p_m$ and $prob_{\text{elite}}$
3: $P :=$ GenerateInitialPopulation($p_{\text{size}}$)
4: Evaluate($P$)
5: while computation time limit not reached do
6:     $P_e :=$ EliteSolutions($P, p_e$)
7:     $P_m :=$ Mutants($P, p_m$)
8:     $P_c :=$ Crossover($P, P_e, prob_{\text{elite}}$)
9:     Evaluate($P_m \cup P_c$)                    {NOTE: $P_e$ is already evaluated}
10:    $P := P_e \cup P_m \cup P_c$
11: end while
12: output: Best solution in $P$
```

be less than 0.5, the probability that a given elite individual is selected for mating $(1/(p_{\text{size}} \cdot p_e))$ is greater than that of a given non-elite individual $(1/(p_{\text{size}}(1 - p_e))$. Therefore, the given elite individual is more likely to pass on its genetic information to forthcoming generations.

4. After generating the offspring and mutant populations (P_c and P_m respectively), both are evaluated in Evaluate($P_m \cup P_c$). Note that the individuals in P_e were already evaluated in previous generations.

5. Finally, the population of the next generation is determined to be the union of P_e with P_m and P_c.

The algorithm terminates when some computation time limit is reached.

3.2.4 Evaluation of an Individual

As we previously mentioned, a decoder is used to map solutions from the hypercube to solutions in the solution space, where fitness is computed.

In the following sections, we propose three different decoders for the problem. In all our versions, individuals are represented as chromosomes of size $n = |V|$, where V is the set of vertices of the input graph. Furthermore, we will consider an individual $\pi = \pi_1, \pi_2, ..., \pi_n$, where $\pi_i \in [0, 1]$.

Decoder 1

In this first implementation, the decoder generates a permutation of vertices of the input graph in order to obtain an independent and dominating set S.

Given an individual π, we translate it into a permutation P of vertices $P = p_1, p_2, ..., p_n$, where $p_i \in \{1, |V|\}$. .

In this case, the ith position of P corresponds to the ith position of π, and the jth lowest value in π corresponds to vertex j in P. More formally, let $\ell_j(\pi)$ be the index of the jth lowest value in π. Then, $p_i := j$, where $i = \ell_j(\pi)$. Figure 3.5 shows an example of this process.

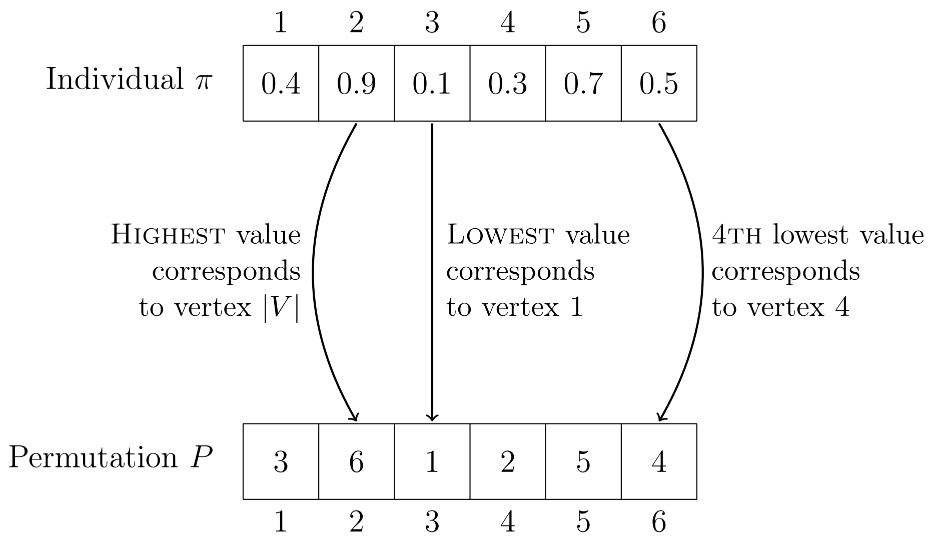

1
2
3
4
5
6
Individual π
0.4
0.9
0.1
0.3
0.7
0.5
HIGHEST value corresponds to vertex $|V|$
LOWEST value corresponds to vertex 1
4TH lowest value corresponds to vertex 4
Permutation P
3
6
1
2
5
4
1
2
3
4
5
6

Figure 3.5: Decoding process of an individual (version 1).

Once this permutation is obtained, we need to construct the solution S. To do so, we start with an empty solution $S := \emptyset$. At each iteration, we add the first vertex v from P to the solution S ($S := S \cup v$) and remove $N_{G'}(v)$ from the permutation. We proceed until we get to the empty set $P = \emptyset$. Note that S is now an independent and dominating set in G.

The only thing left is to calculate the fitness of the individual using the objective function previously stated in Equation 1.1.

Decoder 2

This second decoder also generates a permutation of vertices, but it proceeds in a different way.

In this case, the value of the nth position of π states the "priority" of vertex n in the permutation. More formally, let $\mathcal{L}_j(\pi)$ be the index of the jth largest value in π. Then, $P_j := i$, where $i = \mathcal{L}_j(\pi)$. Figure 3.6 shows an example of this process.

From this point, we proceed by obtaining the solution S and calculating the fitness in the same way as in the previous decoder.

Decoder 3

In this third decoder, the process involves the application of the two previously mentioned greedy heuristics (see Section 3.1), using the values of the individual to bias the original

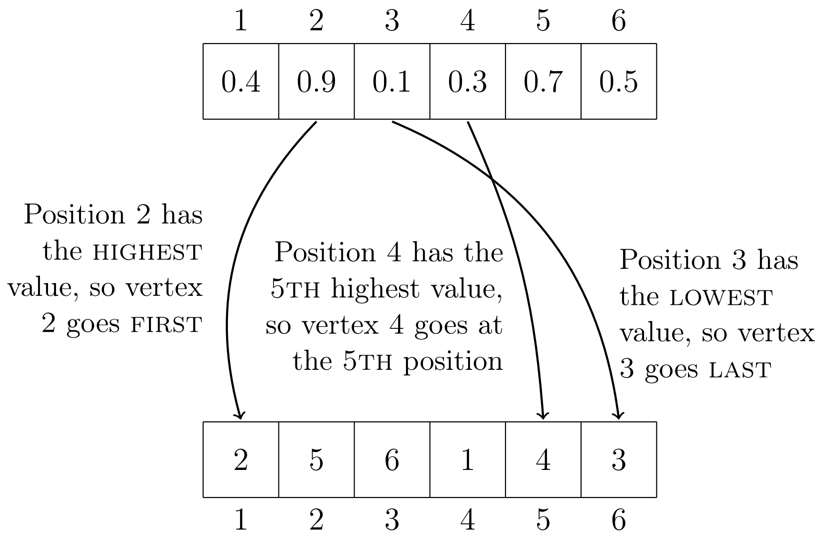

1
2
3
4
5
6
0.4
0.9
0.1
0.3
0.7
0.5
Position 2 has the HIGHEST value, so vertex 2 goes FIRST
Position 4 has the 5TH highest value, so vertex 4 goes at the 5TH position
Position 3 has the LOWEST value, so vertex 3 goes LAST
2
5
6
1
4
3
1
2
3
4
5
6

Figure 3.6: Decoding process of an individual (version 2).

greedy function weights.

However, the following changes have to be applied to the original greedy functions in order to implement this method:

- Greedy function η_1' of Greedy1.

  Greedy function η_1 is replaced by the following:

  $$\eta_1'(v_i) := \frac{deg_{G'}(v_i) + 1}{w(v_i)} \cdot \pi_i$$

  Note that we add 1 to the numerator so the allele can still influence the result when $deg_{G'}(v_i) = 0$.

  Greedy1 is then applied using the new function η_1'. We will denote the the objective function value of the solution produced by Greedy1 on the basis of π as $f_1(\pi)$.

- Greedy function η_2' of Greedy2.

  A similar procedure is used for the evaluation of an individual by means of Greedy2. However, Greedy2 differs from Greedy1 in the way it selects the vertex at each iteration, choosing the one with the lowest greedy function value instead of the largest. If we proceeded as in Greedy1 (by simply multiplying the original greedy function by the value of the individual), the same value of π would affect oppositely on both functions. To fix this, we simply multiply η_2 by -1.

  Nonetheless, we are now facing with a different problem, as η_2 now actually produces negative values. For this reason, when given a partial solution S, we first calculate:

  $$\eta_2^{\max} := \max\{\eta_2(v) \mid v \in V'\}$$

Then, η_2 is replaced by the following greedy function:

$$\eta_2'(v_i) := (-\eta_2(v_i) + \eta_2^{\max} + 1) \cdot \pi_i$$

Note that we also add 1 to the function's value (for the same reason as before). We will denote the objective function value of the solution produced by GREEDY2 on the basis of an individual π as $f_2(\pi)$.

Then, for the evaluation of an individual π, and once these changes are applied to the original greedy functions, our BRKGA assigns to an individual π the value $\min\{f_1(\pi), f_2(\pi)\}$.

3.2.5 Rollout mechanism

By observing the good performance of the rollout mechanism on this problem, regarding its application on both greedy algorithms, we decided to apply it to the three different decoders designed for the genetic algorithm. This technique seems to work well when stuck in a local minima, as it explores the search space around the individual. In this problem, this phenomenon is more accentuated, as choosing one vertex over another can drastically change the solution due to the property of the set being independent.

The rollout applies in a similar way as before (see Section 3.1.1). However, the temporal cost of the mechanism is rather high, so applying it to the evaluation of the whole population would be extremely time consuming and would make the population not able to evolve over sufficient generations. For this purpose, the following three parameters were added to the algorithm in order to control its usage:

1. r_{size}. Stands for *rollout size*, and controls the amplitude of the application. More specifically, in each iteration, instead of calculating the final objective function obtained by adding each one of the possible vertices to the solution, it is only calculated for the r_{size} nodes with the best value (according to the greedy function or other means, depending on the decoder).

2. r_{portion}. Stands for *rollout portion*, and indicates the percentage of the crossover population to which the mechanism is applied. In order to shorten the duration of a generation, we decided it is not necessary to evaluate all of the individuals this way. To this end, we decided to discard elite and mutant individuals, as the former are already evaluated and the latter can be in any point of the search space, so they are not of great interest. The rollout mechanism, then, is only used on a portion of the offspring individuals.

3. r_{stop}. Stands for *rollout stop* and determines the percentage of vertices added to the solution (or discarded) at which the rollout is no longer used. Recall that, at each iteration of the solution construction, the mechanism is applied in order to choose which vertex is added to the set. It is clear to see that, when there are only few

vertices remaining, the mechanism effectiveness decreases, as little to non variation occurs when selecting one vertex over another. Consequently, the rollout can be stopped earlier in order to reduce resource consumption.

Decoders 1 & 2

The mechanism is applied in the same way in both, once the permutation P of vertices has been created. Recall that the solution S is constructed by adding, at each step, the front element from P and removing the vertices adjacent to it from the permutation. This is repeated until P is empty.

With the application of the rollout mechanism, we instead proceed this way.

Let $size := min\{r_{\text{size}}, |P|\}$ and $P[size]$ be the first $size$ elements from permutation P. Given a vertex $v \in P$, we denominate the *projection of v under P* ($p(v)$) to the objective function value of the final solution obtained if we moved vertex v to the front of the permutation and proceeded as normal (until P is empty).

Then, at each iteration, we take the vertex $v \in P[size]$ with the lowest $p(v)$ value, add it to the solution S and remove the vertices in the closed neighborhood of v from the permutation. This process is repeated until $|P| <= (1 - r_{\text{stop}}) \cdot |V|$ (meaning that the r_{stop} of the vertices have either been chosen or discarded). At this point, subsequent iterations are completed as in the standard decoder versions.

The pseudo-code of this procedure can be found in Algorithm 6.

Algorithm 6 Pseudo-code of the rollout implementation of decoders 1 and 2.

```
1: input: an undirected weighted graph $G = (V, E, w)$
2: input: a permutation of vertices $P$
3: input: parameter values for $r_{\text{size}}$ and $r_{\text{stop}}$    {NOTE: $r_{\text{portion}}$ is used outside the decoding process}
4: $S := \emptyset$
5: while $P \neq \emptyset$ do
6:    if $|P| > (1 - r_{\text{stop}}) \cdot |V|$ then
7:       $size := min\{r_{\text{size}}, |P|\}$
8:       $v^* := \text{argmin}\{p(v) \mid v \in P[size]\}$
9:    else
10:      $v^* := p_0$    {NOTE: $p_0$ is the first element in P}
11:   end if
12:   $S := S \cup \{v^*\}$
13:   Remove from $P$ all nodes from $N[v^*]$
14: end while
15: output: An independent dominating set $S$ of $G$
```

Decoder 3

In this version, the rollout mechanism is applied to the two greedy algorithms used to evaluate an individual in the same way as in their standalone version (see Section 3.1.1). It is obvious, though, that we are using the modified greedy functions instead.

However, we have to take into account the new parameters. First, the rollout mechanism is only applied until $V' <= (1 - r_{\text{stop}}) \cdot |V|$. Second, only the r_{size} vertices with the best greedy function value are considered when selecting the one with the lowest projection value.

Chapter 4

Experimental evaluation

In this chapter, we present the experimental evaluation of our algorithms. First, the set of benchmark instances is described. Then, an explanation of the tuning process of our algorithms is provided. Finally, the results obtained are presented and discussed.

The following ten algorithmic approaches have been considered: The two greedy rollout versions (1) Greedy1-Rollout and (2) Greedy2-Rollout; the greedy multistart approaches (3) Greedy1-Multistart and (4) Greedy2-Multistart; the BRKGA with its three different decoders (5) Brkga-v1, (6) Brkga-v2 and (7) Brkga-v3 and their respective rollout versions (8) Brkga-v1-Rollout, (9) Brkga-v2-Rollout and (10) Brkga-v3-Rollout.

The algorithms were implemented in ANSI C++, using GCC 7.4.0 for compiling the code. The experimental evaluation was performed on a cluster of computers with "Intel® Xeon® CPU 5670" CPUs of 12 nuclei of 2933 MHz and (in total) 32 Gigabytes of RAM.

4.1 Benchmark instances

For a fair experimental evaluation of our algorithms, we considered the same problem instances as in [23], generated as described in the following.

Graphs are divided into two types: (1) *random graphs* and (2) *random geometric graphs*, both of different properties (density, size, etc.). More specifically, for each type, graphs of order $|V| \in \{100, 500, 1000\}$ were generated. In addition, three different schemes for generating node and edge weights were considered:

1. *Neutral graphs*: Both node and edge weights are integers drawn uniformly at random from {0...100}.

2. *Node-oriented graphs*: Node weights are integers w chosen uniformly at random from {0..1000}, while edge weights were chosen uniformly at random from the interval {0...100}. Note that, in these graphs, node weights directly impact the choice of nodes themselves.

3. *Edge-oriented graphs*: Edge weights are integers w chosen uniformly at random from $\{0..1000\}$, while node weights w are chosen uniformly at random from the interval $\{0...100\}$. Note that, in this case, the weights connecting non-chosen nodes to chosen ones greatly influence the selection of nodes.

For the density, a different approach was used for each type of graph:

1. *Random graphs*: In these graphs, any two nodes may be connected. To control the density, edges were generated between any pair of nodes with probability $ep \in \{0.05, 0.15, 0.25\}$.

2. *Random geometric graphs*: Here, only nodes that are placed close together may be connected. Graphs were created as follows. First, $|V|$ nodes were assigned random coordinates from the unit square. Then, each pair of nodes at a distance smaller or equal than a given radius r were connected. To produce graphs with similar densities than the random graphs the settings for r were as follows: $r \in \{0.14, 0.24, 0.34\}$.

For each combination of graph type, number of nodes, edge probability (or radius) and weight generation scheme, 10 problem instances were generated, resulting in a total of 540 graphs: 270 for each type (random graphs and random geometric graphs).

4.2 Tuning

Before evaluating our approaches, an adjustment of the algorithms' parameters was needed. For this purpose, the `irace` tool was used [20]. Its main purpose is to automatically configure optimization algorithms by finding the most appropriate settings given a set of problem instances. In the following subsections, we describe the tuning setups and results.

4.2.1 General considerations

We applied the tuning to each algorithm with parameters to configure: the two greedy multistart algorithms, the three different versions of the BRKGA and their respective rollout implementations. This makes a total of 8 algorithms to tune.

For each one, the `irace` tool was applied 3 times (one for each graph order $|V|$). For each tuning run, 18 different instances were used, selecting the first one of the 10 available instances for each combination of type of graph, weight generation scheme and edge probability (or radius).

BRKGA rollout tunings were given a budget of 2000 applications of the algorithm per tuning run, while the others were given a maximum of 1000 applications. This was decided because the former algorithms have more parameters to adjust than the latter ones. For each application, a time limit of $3 \cdot |V|$ CPU seconds was given.

4.2.2 Tuning of the multistarts algorithms

The following parameters and ranges were considered:

- $d_{\text{rate}} \in \{0.5, 0.6, 0.7, 0.8, 0.9\}$
- $l_{\text{size}} \in \{2, 3, 5, 10, 20\}$

Results obtained for both the multistart versions of GREEDY1 and of GREEDY2 are shown in Table 4.1 and Table 4.2, respectively.

Table 4.1: Results of tuning GREEDY1-MULTISTART

$\|V\|$	d_{rate}	l_{size}
100	0.5	20
500	0.5	20
1000	0.6	20

Table 4.2: Results of tuning GREEDY2-MULTISTART

$\|V\|$	d_{rate}	l_{size}
100	0.5	20
500	0.5	10
1000	0.6	10

The results show that a lower d_{rate} and a higher l_{size} are preferred. In addition, both greedy algorithms tend to become slightly more deterministic as the order of the graphs increases.

4.2.3 Tuning of BRKGAs

For the BRKGA, four parameters were considered for tuning, with the following value ranges:

- $p_{\text{size}} \in \{10, 20, 50, 100, 200, 500\}$
- $p_e \in \{0.05, 0.1, 0.15, 0.2, 0.25\}$
- $p_m \in \{0.1, 0.15, 0.2, 0.25, 0.3\}$
- $prob_{\text{elite}} \in \{0.5, 0.6, 0.7, 0.8, 0.9\}$

Note that the ranges of the last three parameters correspond to the ones recommended in [15].

Results obtained from the tuning process are presented in Table 4.3 (VERSION 1), Table 4.4 (VERSION 2) and Table 4.5 (VERSION 3).

In this case, however, the trends differ greatly among versions.

In the first version, the population size (p_{size}) is rather low for versions 2 and 3. The size of the mutant population (p_m) starts at the minimum value and increases with the graph order. The size of the elite population (p_e) and the elite allele inheritance probability ($prob_{\text{elite}}$) values are quite average, with the latter being slightly lower in the case $|V| = 500$.

Table 4.3: Results of tuning BRKGA-V1

$\lvert V\rvert$	p_{size}	p_e	p_m	$prob_{\text{elite}}$
100	100	0.15	0.1	0.8
500	20	0.2	0.2	0.6
1000	50	0.1	0.3	0.8

Concerning the second version, the trends seem to be similar regardless of the order of the graph. A high p_{size}, p_e and p_m resulted best in all three cases.

Table 4.4: Results of tuning BRKGA-V2

$\lvert V\rvert$	p_{size}	p_e	p_m	$prob_{\text{elite}}$
100	500	0.25	0.3	0.8
500	500	0.2	0.3	0.6
1000	500	0.25	0.3	0.7

Finally, the results of the third version show that, for graphs of small order, a small population is preferred. Otherwise, big populations are chosen. The mutant population is rather large, and increases with $|V|$, along with $prob_{\text{elite}}$. On the contrary, the elite population size decreases.

Table 4.5: Results of tuning BRKGA-V3

$\lvert V\rvert$	p_{size}	p_e	p_m	$prob_{\text{elite}}$
100	20	0.2	0.2	0.6
500	500	0.15	0.2	0.6
1000	500	0.1	0.25	0.7

4.2.4 Tuning of BRKGAs rollout version

In addition to the five parameters of the default BRKGA versions, the following three parameters (with their respective value ranges) must be tuned in the case of the rollout versions:

- $r_{\text{size}} \in \{2, 5, 10, 20\}$
- $r_{\text{portion}} \in \{0.25, 0.5, 0.75, 1.0\}$

- $r_{\text{stop}} \in \{0.25, 0.5, 0.75, 1.0\}$

As in the three BRKGA standard approaches, the trends are distinct depending on the algorithm version.

Concerning the first version, the results, presented in Table 4.6, show that the size of the population must be kept low (except when the graph has a small order). Interestingly, when the rollout size (r_{size}) increases, r_{portion} and r_{stop} seem to follow the same trend, while the elite population size (p_e) tends to decrease. In addition, $prob_{\text{elite}}$ is minimum when the graph order is 100. Other parameters have intermediate values.

Table 4.6: Results of tuning BRKGA-V1-ROLLOUT

$\|V\|$	p_{size}	p_e	p_m	$prob_{\text{elite}}$	r_{size}	r_{portion}	r_{stop}
100	100	0.15	0.15	0.5	5	0.25	0.25
500	20	0.05	0.2	0.7	20	0.75	1.0
1000	20	0.25	0.1	0.7	2	0.25	0.5

Table 4.7 shows the tuning results for the second version. In general, large values for the population and rollout size are preferred. Note that one slightly increases when the other decreases. The elite allele inheritance probability is kept low. On the contrary, the size of the elite and mutant populations must be rather high (with the exception of p_e when $|V| = 100$). Finally, the r_{portion} value is inversely proportional to the order of the graph.

Table 4.7: Results of tuning BRKGA-V2-ROLLOUT

$\|V\|$	p_{size}	p_e	p_m	$prob_{\text{elite}}$	r_{size}	r_{portion}	r_{stop}
100	100	0.05	0.3	0.5	20	1.0	0.75
500	500	0.2	0.3	0.5	10	0.75	1.0
1000	500	0.25	0.25	0.6	10	0.5	0.25

The third version seems to have a clear trend. The size of the rollout (r_{size}) and the fraction of the offspring population to which the rollout is applied (r_{portion}) increase with the order of the graph, whilst the population size, the size of the elite population and the elite allele inheritance probability ($prob_{\text{elite}}$) decrease. This makes a lot of sense as, the larger the rollout size, the more time it takes to evaluate an individual, so the population must be smaller in order to maintain a reasonable duration of an iteration. Other parameters, such as p_m and r_{stop}, are rather high. These results are shown in Table 4.8

Table 4.8: Results of tuning BRKGA-V3-ROLLOUT

$\|V\|$	p_{size}	p_e	p_m	$prob_{\text{elite}}$	r_{size}	r_{portion}	r_{stop}
100	500	0.25	0.25	0.9	2	0.25	0.75
500	50	0.2	0.3	0.6	5	0.5	1.0
1000	20	0.1	0.2	0.5	20	0.75	1.0

4.3 Results

All the implemented algorithms have been applied once to each problem instance, with a maximum computation time of $3 \cdot |V|$ seconds for all of the applications (except for Greedy1-Rollout and Greedy2-Rollout, which do not have a time limit).

For the evaluation of the results, the algorithms were grouped in two: greedy algorithms and genetic algorithms. The forthcoming tables follow the same format as in [23] for the sake of a clear comparison. The first three columns indicate the order of the graph ($|V|$), the weight generation scheme (N for neutral graphs, V for vertex-oriented and E for edge-oriented) and the graph density (indicated by the edge probability ep for random graphs and the radius r for random geometric graphs).

The results for each algorithm are presented in two columns. The first one (labeled *result*) indicates the average result obtained by applying the algorithm to the 10 corresponding instances. The best result for each row is marked with a gray background. The second column (labeled *time*) has distinct readings. In the case of Greedy1-Rollout and Greedy2-Rollout, it indicates the average computation time in seconds. For the others, it illustrates the average time at which the best solutions of each application were encountered.

Notched boxplots were generated to offer a more visual representation of the algorithms' performance (Figures 4.1 to 4.6). These are divided by graph type and weight generation scheme.

Tables 4.9 and 4.10 contain the experimental results for the Greedy approaches concerning random graphs and random geometric graphs, respectively. The following analysis can be made by examining them:

- The multistart version of Greedy2 is clearly the one which performs best. It is followed by Greedy2-Rollout, specially in high order graphs. In general, the versions using the second greedy heuristic work better than those using the first one, even though the latter ones can compete in the case of node-oriented graphs.

- Concerning the two types of graphs (random graphs and random geometric graphs), no major differences can be observed regarding the performance of our algorithms. Greedy2-Rollout, however, seems to work slightly better in random geometric graphs of higher order.

- Greedy2-Rollout outperforms Greedy1-Rollout except when nodes have higher weights than edges. This can be observed in Figure 4.2. This does not apply to the multistart versions, as Greedy2-Multistart obtains better results than its analogous version Greedy1-Multistart in every single run.

- Concerning computation time, both greedy rollout versions are by far the fastest ones of the new approaches. Greedy1-Rollout tends to take more time as graphs become denser, while Greedy2-Rollout has an opposite tendency.

Table 4.9: Numerical results for random graphs. Greedy algorithms.

$\|V\|$	Weight scheme	*ep*	Greedy1		Greedy2		Greedy1-Rollout		Greedy2-Rollout		Greedy1-Multistart		Greedy2-Multistart	
			result	**time**	**result**	**time**	**result**	**time**	**result**	**time**	**result**	**time**	**result**	**time**
100	N	0.05	3589.1	<0.1	3519.1	<0.1	3103.7	0.1	3110.0	0.1	3096.6	120.9	3069.5	114.9
		0.15	3014.4	<0.1	2981.3	<0.1	2454.9	0.1	2469.1	0.1	2386.4	84.1	2343.7	83.9
		0.25	2883.5	<0.1	2796.1	<0.1	2224.3	0.1	2193.2	<0.1	2100.2	26.2	2072.8	37.4
	V	0.05	10465.6	<0.1	11756.6	<0.1	8617.0	0.1	8484.2	0.1	8204.7	102.0	8322.5	122.7
		0.15	4891.6	<0.1	5845.4	<0.1	3261.2	0.1	3577.4	0.1	3290.3	66.7	3098.3	69.6
		0.25	3297.5	<0.1	3488.9	<0.1	2001.6	<0.1	2102.3	<0.1	1899.8	9.8	1808.4	3.9
	E	0.05	25698.7	<0.1	22269.3	<0.1	16792.5	0.1	15519.4	0.1	16108.4	204.0	15103.5	133.5
		0.15	27528.4	<0.1	23404.5	<0.1	17643.7	0.1	16819.0	<0.1	15590.5	140.5	14927.2	69.2
		0.25	25451.4	<0.1	21770.0	<0.1	18317.6	0.1	16408.8	<0.1	14768.3	71.7	14433.5	68.6
500	N	0.05	14143.1	<0.1	13535.1	<0.1	11335.0	13.5	11375.2	13.6	12114.3	817.7	11617.6	703.2
		0.15	12268.5	<0.1	11558.0	<0.1	9257.8	17.6	9039.4	8.9	9391.1	873.3	9057.3	697.3
		0.25	11630.3	<0.1	10429.5	0.1	8604.5	18.2	8605.8	5.0	8413.1	661.4	8347.7	858.8
	V	0.05	15501.5	<0.1	18298.1	<0.1	11451.8	10.7	12041.5	18.9	13845.2	855.6	11977.5	469.1
		0.15	6496.3	<0.1	7300.1	<0.1	4256.6	10.2	4340.2	11.5	4728.8	553.1	3817.2	625.1
		0.25	4212.4	<0.1	4463.7	0.1	2897.1	14.0	2955.5	5.7	2956.7	610.4	2625.7	439.0
	E	0.05	125357.6	<0.1	108178.0	<0.1	89641.2	9.0	83385.9	13.4	93272.3	776.7	87558.8	776.8
		0.15	114951.0	<0.1	102365.1	<0.1	82034.5	15.1	76928.6	6.7	80687.3	624.1	77037.2	644.3
		0.25	111012.3	<0.1	99018.2	0.1	79853.7	15.8	73859.3	7.3	75170.1	625.9	71911.0	945.2
1000	N	0.05	25569.6	<0.1	23489.7	0.1	20553.0	115.1	20146.6	179.7	21671.9	1730.8	20750.8	1580.0
		0.15	20827.1	<0.1	20689.1	0.2	16773.7	181.2	16703.5	143.7	17132.7	1352.8	16827.7	1176.4
		0.25	20858.8	<0.1	19280.5	0.4	15686.8	237.0	15451.8	113.2	15920.6	1616.6	15273.9	1459.5
	V	0.05	18048.6	<0.1	20142.3	0.1	12470.7	70.0	13841.4	203.1	16685.7	1261.5	13980.9	1404.6
		0.15	7408.3	<0.1	7987.4	0.2	5253.5	175.1	5364.0	138.2	5863.1	1212.0	5045.6	1846.8
		0.25	4941.9	<0.1	5566.6	0.4	3861.3	202.0	3930.3	122.3	4162.0	1206.9	3684.7	1127.1
	E	0.05	238600.0	<0.1	202992.0	0.1	174265.9	155.9	162348.2	141.8	189115.1	1650.4	172147.2	2025.2
		0.15	209709.3	<0.1	182726.6	0.2	157623.7	162.7	145440.1	129.6	160814.5	2013.4	147968.6	1289.0
		0.25	198537.0	<0.1	181150.0	0.4	150418.8	207.0	141696.5	121.5	149083.2	1465.5	137730.8	1585.9

Table 4.10: Numerical results for random geometric graphs. Greedy algorithms.

$\|V\|$	Weight scheme	*ep*	Greedy1		Greedy2		Greedy1-Rollout		Greedy2-Rollout		Greedy1-Multistart		Greedy2-Multistart	
			result	**time**	**result**	**time**	**result**	**time**	**result**	**time**	**result**	**time**	**result**	**time**
100	N	0.05	3870.6	<0.1	3464.4	<0.1	3338.6	0.1	3275.1	<0.1	3407.3	158.9	3269.3	115.1
		0.15	3798.8	<0.1	3378.1	<0.1	3043.3	0.1	2944.1	<0.1	2980.8	168.9	2887.4	56.0
		0.25	3766.6	<0.1	3388.1	<0.1	2939.2	0.1	2908.1	<0.1	2905.0	103.7	2832.9	16.8
	V	0.05	7364.9	<0.1	7514.3	<0.1	5732.8	0.1	5802.2	0.1	6196.7	70.3	5809.2	109.0
		0.15	2880.1	<0.1	2724.2	<0.1	2041.8	0.1	2056.2	<0.1	2071.8	36.7	1981.8	4.7
		0.25	1741.0	<0.1	1832.3	<0.1	1034.4	0.1	1045.1	<0.1	966.1	3.0	940.5	0.5
	E	0.05	29011.6	<0.1	24998.3	<0.1	20156.4	<0.1	19639.2	0.1	21647.3	102.6	19623.4	145.6
		0.15	35312.1	<0.1	28647.1	<0.1	24712.0	0.1	23225.2	<0.1	23813.8	181.8	22214.3	92.7
		0.25	37929.4	<0.1	30503.4	<0.1	26016.7	0.1	24613.6	<0.1	24769.2	164.8	23759.2	17.2
500	N	0.05	18408.3	<0.1	16208.4	<0.1	14518.1	8.0	13956.2	8.8	16014.6	658.5	14704.8	757.4
		0.15	18548.8	<0.1	15882.9	<0.1	13804.3	11.6	13572.5	5.0	15037.3	749.8	13680.4	476.1
		0.25	18311.4	<0.1	15497.8	0.1	14319.7	20.6	13534.8	4.9	14663.2	820.7	13386.7	876.5
	V	0.05	6807.8	<0.1	7087.8	<0.1	4561.2	7.4	4645.4	9.1	6138.7	914.6	4784.4	686.9
		0.15	3526.6	<0.1	3121.1	<0.1	2595.7	13.1	2623.5	6.8	2850.7	443.6	2574.3	403.8
		0.25	2632.2	<0.1	2830.7	0.1	2245.3	18.9	2236.3	5.7	2320.3	408.9	2186.3	110.1
	E	0.05	177816.7	<0.1	148267.0	<0.1	124462.1	9.1	120566.2	8.3	146264.7	656.6	127288.2	665.6
		0.15	181739.2	<0.1	149980.2	<0.1	133659.5	14.2	126999.4	5.7	141560.6	962.0	127036.3	685.9
		0.25	190996.4	<0.1	155128.6	0.1	134416.6	19.8	126556.2	5.0	139187.3	767.0	127007.2	629.9
1000	N	0.05	36214.6	<0.1	32393.4	0.1	27871.0	137.5	27695.4	118.2	32543.2	1313.6	29200.6	1178.2
		0.15	36750.4	<0.1	31462.5	0.2	27760.5	180.8	26555.3	79.4	30758.0	1684.7	27108.6	1025.3
		0.25	36913.2	<0.1	30752.1	0.3	28323.8	222.1	26382.7	59.7	29790.1	1616.9	26395.5	1151.6
	V	0.05	8552.3	<0.1	8613.7	0.1	6017.9	128.9	6131.3	118.1	7354.2	1367.8	6423.3	1476.9
		0.15	4977.4	<0.1	5146.2	0.2	4360.8	182.8	4354.2	47.9	4627.8	1175.8	4335.4	1214.8
		0.25	4656.5	<0.1	4693.1	0.4	4069.3	212.4	4038.4	61.2	4301.1	853.6	4033.7	688.6
	E	0.05	358550.4	<0.1	305034.7	0.1	264892.6	141.4	251377.0	98.2	306044.2	1554.7	269786.1	1735.3
		0.15	357735.8	<0.1	295099.4	0.2	269188.7	199.3	251404.5	89.3	288534.3	1945.7	258242.8	1277.4
		0.25	369051.9	<0.1	303712.9	0.3	271223.5	233.5	252243.0	36.8	286347.3	1490.1	253251.9	1990.6

Table 4.11: Numerical results for random graphs. Genetic algorithms.

$\|V\|$	Weight scheme	*ep*	Brkga-V1		Brkga-V2		Brkga-V3		Brkga-V1-Rollout		Brkga-V2-Rollout		Brkga-V3-Rollout			Best solution	
			result	**time**	**result**	**time**	**result**	**time**	**result**	**time**	**result**	**time**	**result**	**time**		**result**	**algorithm**
100	N	0.05	3055.4	40.9	3052.6	25.1	3049.8	61.3	3053.1	30.8	3049.8	34.0	3050.4	23.0	=	3049.8	Ilp-1|2 Pbig*
		0.15	2333.3	98.6	2334.0	33.0	2336.8	52.0	2333.0	54.8	2330.9	42.5	2333.6	58.9	=	2330.9	Pbig
		0.25	2069.3	20.2	2069.3	13.2	2072.8	28.5	2069.3	1.9	2069.3	3.7	2082.0	8.2	<	2070.9	Pbig
	V	0.05	8030.8	73.7	7739.6	2.8	7715.4	21.9	8077.2	83.0	7772.1	11.8	7715.4	21.8	=	7715.4	Ilp*
		0.15	3163.0	90.1	3180.7	6.9	3046.6	55.7	3158.9	88.6	3188.0	67.3	3058.5	38.7	=	3046.6	Ilp*
		0.25	1808.4	76.1	1825.7	12.4	1808.4	1.1	1826.6	46.4	1813.3	15.4	1808.4	1.1	=	1808.4	Ilp* Pbig*
	E	0.05	14407.2	58.8	14378.7	36.4	14425.9	23.3	14378.7	48.6	14405.4	21.5	14427.2	31.9	=	14378.7	Ilp* Pbig*
		0.15	14563.3	33.0	14563.3	<0.1	14753.6	89.7	14563.3	47.8	14563.3	15.7	14797.0	18.4	=	14563.3	Cmsa-Pbig
		0.25	14382.2	15.7	14382.2	0.4	14497.8	30.6	14382.2	5.3	14382.2	2.9	14519.3	49.2	=	14382.2	Cmsa-Pbig
500	N	0.05	11476.1	983.6	10348.6	484.2	10773.4	735.9	11383.4	1104.3	10288.2	410.2	11062.7	1267.9	>	10140.6	Cmsa-Pbig
		0.15	8931.2	639.3	8330.7	369.8	8733.1	978.2	8961.1	901.5	8287.2	563.0	8673.7	901.9	>	8046.2	Cmsa-Pbig
		0.25	8206.1	483.7	7773.7	364.7	8147.9	873.9	8187.8	814.0	7733.0	667.6	7992.6	771.0	>	7433.0	Cmsa-Pbig
	V	0.05	16467.7	1287.3	10778.9	257.3	9750.3	326.9	15854.8	1187.2	10479.1	545.7	10011.6	876.5	>	9588.2	Cmsa-Pbig
		0.15	6485.2	807.0	4099.6	427.4	3725.7	349.3	6381.6	929.4	4039.8	620.6	3743.0	550.5	>	3557.8	Cmsa-Pbig
		0.25	4039.5	678.1	2978.9	426.5	2618.9	300.8	4051.7	858.5	2797.4	545.4	2610.2	452.6	>	2586.5	Cmsa-Pbig
	E	0.05	78375.2	1143.6	69710.4	365.7	74519.3	990.4	78425.9	1048.5	68177.5	415.8	79929.0	1184.1	>	67528.9	Cmsa-Pbig
		0.15	72590.4	545.7	66174.9	578.7	71559.0	1001.1	71227.4	843.6	65426.8	720.9	71701.1	951.4	>	62950.1	Cmsa-Pbig
		0.25	68753.0	863.6	64686.3	734.4	68390.8	980.9	69104.4	558.3	64052.4	623.2	69674.7	984.8	>	61411.1	Cmsa-Pbig
1000	N	0.05	20783.6	2091.3	18282.0	1169.4	19572.7	2261.3	20562.6	2045.1	17807.8	1127.2	20231.3	1800.2	>	17723.7	Pbig
		0.15	16729.0	1893.5	15318.2	1873.1	16314.3	1516.4	16516.1	1311.6	16047.9	2089.6	15940.8	1731.1	>	14461.9	Cmsa-Pbig
		0.25	15067.8	1467.9	14666.2	2299.0	15276.9	2012.5	15321.0	1587.9	14898.7	2147.5	14793.8	1272.3	>	13695.6	Cmsa-Pbig
	V	0.05	24388.7	2210.5	12777.9	1627.0	11397.2	985.4	23977.3	2565.8	12673.5	1414.3	12449.2	1727.5	>	11034.6	Cmsa-Pbig
		0.15	9351.4	1587.7	5926.2	1444.9	4875.5	1061.7	9387.9	1306.7	5599.8	1324.7	4778.8	1318.4	>	4456.4	Cmsa-Pbig
		0.25	5910.3	1480.1	4334.6	1351.5	3712.3	1602.9	5867.7	1553.8	4092.4	1949.4	3547.5	1272.2	>	3460.4	Cmsa-Pbig
	E	0.05	161501.4	1732.8	139409.5	1476.8	159883.5	2668.4	160713.8	2489.3	140729.2	2423.9	163567.2	1557.1	>	130889.2	Cmsa-Pbig
		0.15	144906.0	1598.3	132871.7	2060.9	145216.7	2215.7	145117.5	1591.7	143079.7	1846.2	142106.4	1548.9	>	120997.2	Cmsa-Pbig
		0.25	135105.4	1978.2	132215.7	2039.9	136380.9	2040.8	134143.0	1421.1	133227.5	1828.0	135445.6	1845.3	>	120228.7	Cmsa-Pbig

Table 4.12: Numerical results for random geometric graphs. Genetic algorithms.

$\lvert V\rvert$	Weight scheme	*ep*	BRKGA-V1		BRKGA-V2		BRKGA-V3		BRKGA-V1-ROLLOUT		BRKGA-V2-ROLLOUT		BRKGA-V3-ROLLOUT			BEST SOLUTION	
			result	**time**	**result**	**time**	**result**	**time**	**result**	**time**	**result**	**time**	**result**	**time**		**result**	**algorithm**
100	N	0.05	3263.1	68.0	3262.0	0.3	3261.1	13.9	3263.2	49.4	3261.1	2.4	3261.1	12.8	=	3261.1	ILP* PBIG*
		0.15	2891.4	40.3	2885.2	3.1	2882.5	2.3	2886.2	24.4	2883.0	2.2	2882.5	4.0	=	2882.5	PBIG*
		0.25	2828.0	21.5	2828.0	0.4	2832.9	9.0	2828.0	20.6	2828.0	2.7	2832.8	25.1	=	2828.0	PBIG*
	V	0.05	5754.3	111.8	5734.0	4.9	5731.8	4.4	5753.2	27.3	5731.8	10.3	5731.8	10.0	=	5731.8	ILP* PBIG
		0.15	2004.3	77.1	1981.8	0.4	1981.8	0.1	1984.3	28.5	1981.8	0.4	1981.8	1.2	=	1981.8	ILP* PBIG
		0.25	940.5	55.8	940.5	5.4	940.5	0.2	940.5	8.4	940.5	0.8	940.5	0.3	=	940.5	ILP* PBIG*
	E	0.05	19189.4	66.2	19179.4	0.9	19190.0	6.8	19195.0	29.3	19179.4	2.4	19190.0	6.7	=	19179.4	ILP* CMSA-PBIG
		0.15	22090.5	49.0	22065.9	0.9	22081.6	36.7	22066.2	49.9	22065.9	23.1	22090.5	41.0	=	22065.9	ILP-3 CMSA-PBIG
		0.25	23717.6	58.4	23717.6	24.1	23754.8	44.2	23717.6	9.5	23717.6	8.7	23754.8	12.2	=	23717.6	ILP-2 CMSA-PBIG
500	N	0.05	14669.6	1115.9	13441.2	248.9	13495.8	745.9	14760.1	1035.9	13453.9	377.2	13660.0	1249.4	>	13301.2	CMSA-PBIG
		0.15	14001.3	834.4	13022.8	322.2	13205.1	664.5	14032.7	912.1	13094.9	371.1	13043.9	752.8	>	12783.3	CMSA-PBIG
		0.25	13838.1	568.4	13112.5	326.7	13201.7	379.0	13880.2	694.9	13223.5	298.0	13045.1	444.3	>	12954.8	CMSA-PBIG
	V	0.05	6566.5	1235.8	4477.1	450.2	4439.5	312.8	6462.2	1072.7	4459.6	485.1	4399.7	627.4	>	4377.0	ILP-1|2 CMSA-PBIG
		0.15	3395.5	911.6	2599.1	441.1	2574.7	191.3	3359.8	931.3	2589.7	215.9	2573.7	69.1	=	2573.7	PBIG*
		0.25	2497.2	610.3	2208.0	259.9	2182.7	306.6	2527.0	727.3	2225.9	428.4	2181.6	60.1	=	2181.6	PBIG*
	E	0.05	126047.6	999.8	113836.4	295.2	114502.6	516.1	125589.8	1131.7	113226.7	559.1	116180.6	1141.6	>	111638.2	CMSA-PBIG
		0.15	129727.6	998.3	119894.0	546.6	119849.6	679.5	127203.5	1040.2	119202.6	382.7	119508.5	768.7	>	117439.8	CMSA-PBIG
		0.25	128103.7	735.3	122454.0	215.1	123750.2	417.5	127948.8	982.7	122747.7	475.1	122643.7	530.0	>	120883.7	CMSA-PBIG
1000	N	0.05	30013.5	2031.1	26591.6	1652.8	27112.6	1472.7	29897.2	2383.5	26575.4	1703.3	27978.1	1996.2	>	25719.0	CMSA-PBIG
		0.15	28788.1	2027.7	26205.2	1962.9	26250.8	1902.8	28257.2	1860.4	26089.5	1729.3	26331.9	2032.5	>	25138.9	CMSA-PBIG
		0.25	27441.0	2116.0	26245.3	1583.1	25730.2	1254.6	27649.7	1089.8	26091.2	1183.0	25694.5	1764.5	>	25345.0	CMSA-PBIG
	V	0.05	9983.3	2144.8	6033.2	1826.3	5881.5	834.9	10435.6	2642.4	6009.8	1811.3	5940.1	2252.2	>	5869.0	PBIG
		0.15	5947.0	1564.6	4392.4	1738.4	4317.2	795.8	5827.6	1515.2	4443.5	1721.2	4294.1	1049.9	>	4281.2	PBIG
		0.25	4766.6	1609.1	4099.4	1726.5	4005.0	715.6	4656.8	1294.1	4089.2	1798.8	3978.8	495.8	>	3974.3	CMSA-PBIG
	E	0.05	273872.9	2564.2	242345.6	1746.4	247569.1	2073.2	272956.2	2504.8	239555.7	1178.5	254021.3	1607.0	>	233127.8	CMSA-PBIG
		0.15	268228.0	1804.0	246408.7	1265.6	247913.2	1775.3	273781.7	2192.8	245022.2	1357.3	249562.1	2032.7	>	238364.5	CMSA-PBIG
		0.25	263649.7	1834.0	251566.7	1471.1	247597.1	1082.4	263976.2	1598.0	248124.6	1138.5	247054.4	1939.7	>	244685.7	PBIG

Figure 4.1: Notched boxplots for random graphs with a neutral weight scheme.

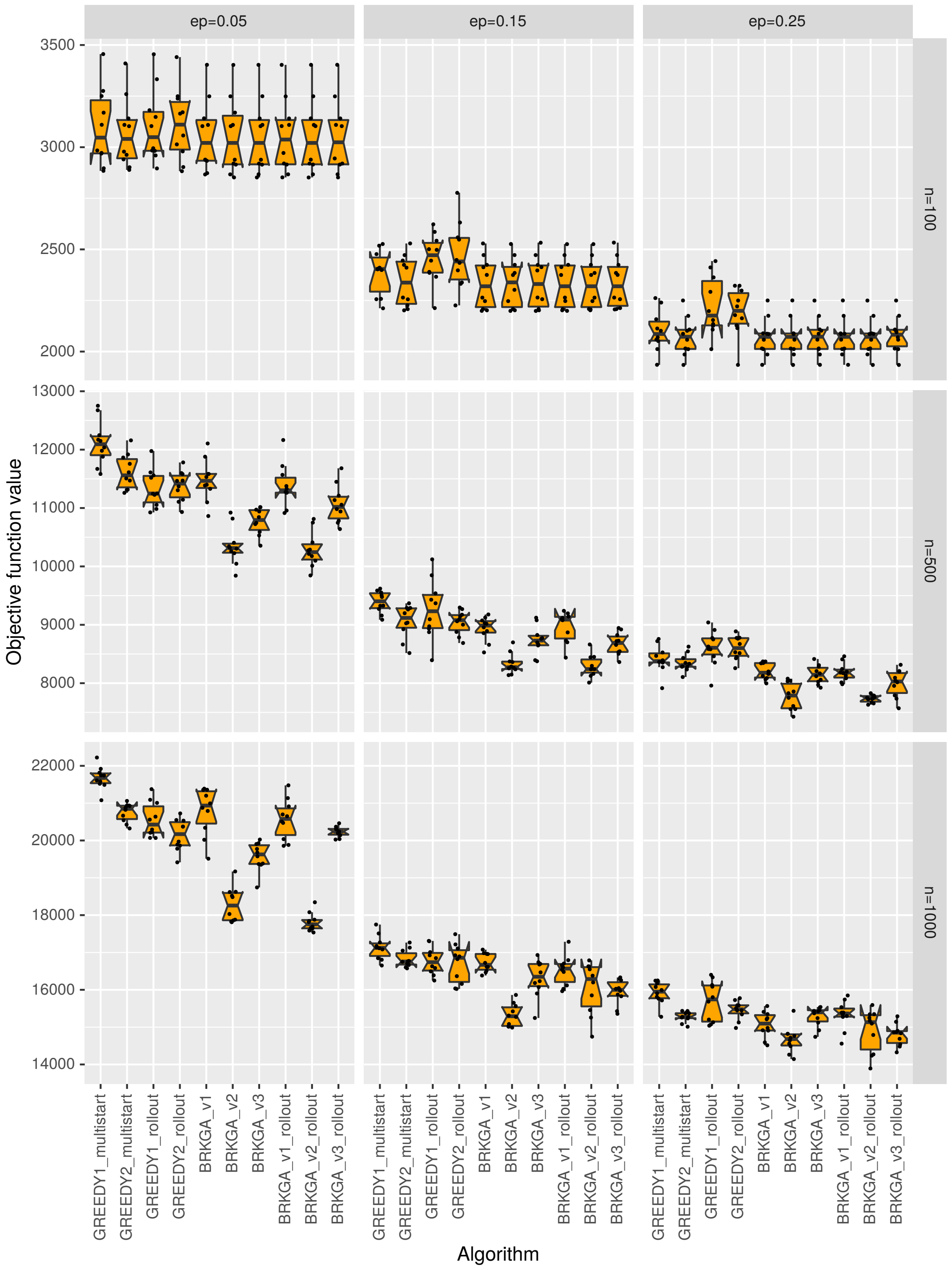


Figure 4.2: Notched boxplots for random graphs with a node-oriented weight scheme.

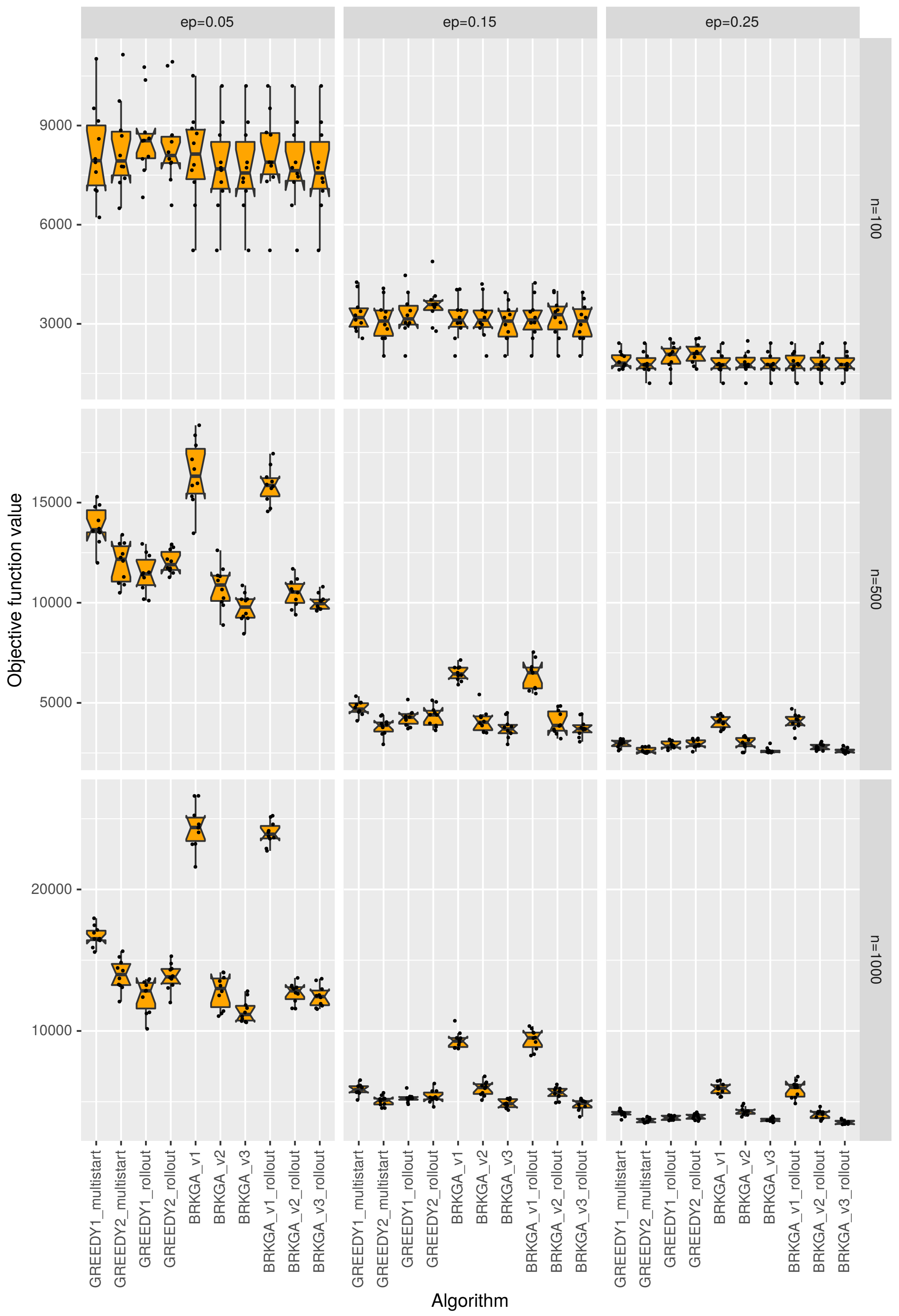


Figure 4.3: Notched boxplots for random graphs with an edge-oriented weight scheme.

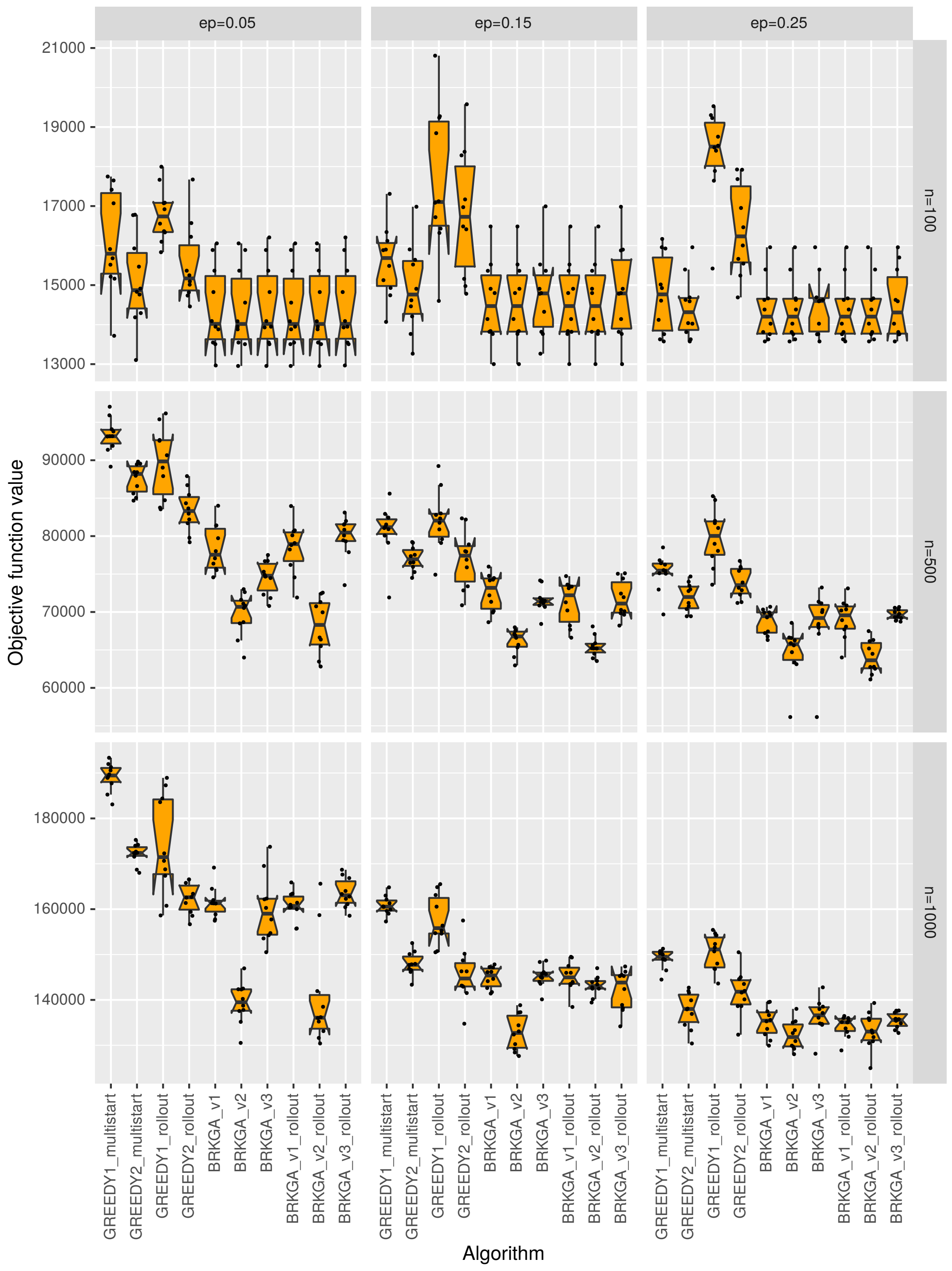


Figure 4.4: Notched boxplots for random geometric graphs with a neutral weight scheme.

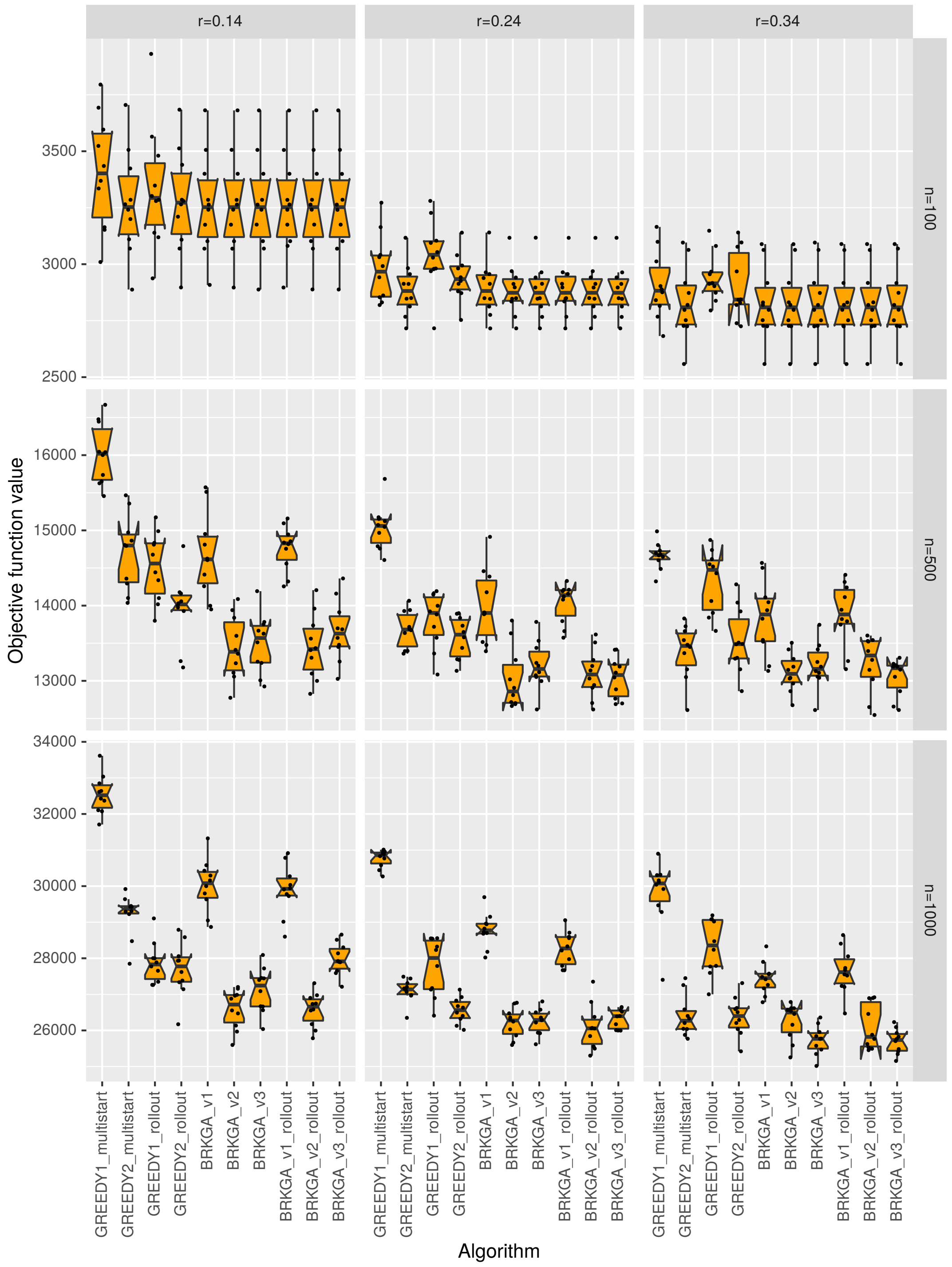


Figure 4.5: Notched boxplots for random geometric graphs with a node-oriented weight scheme.

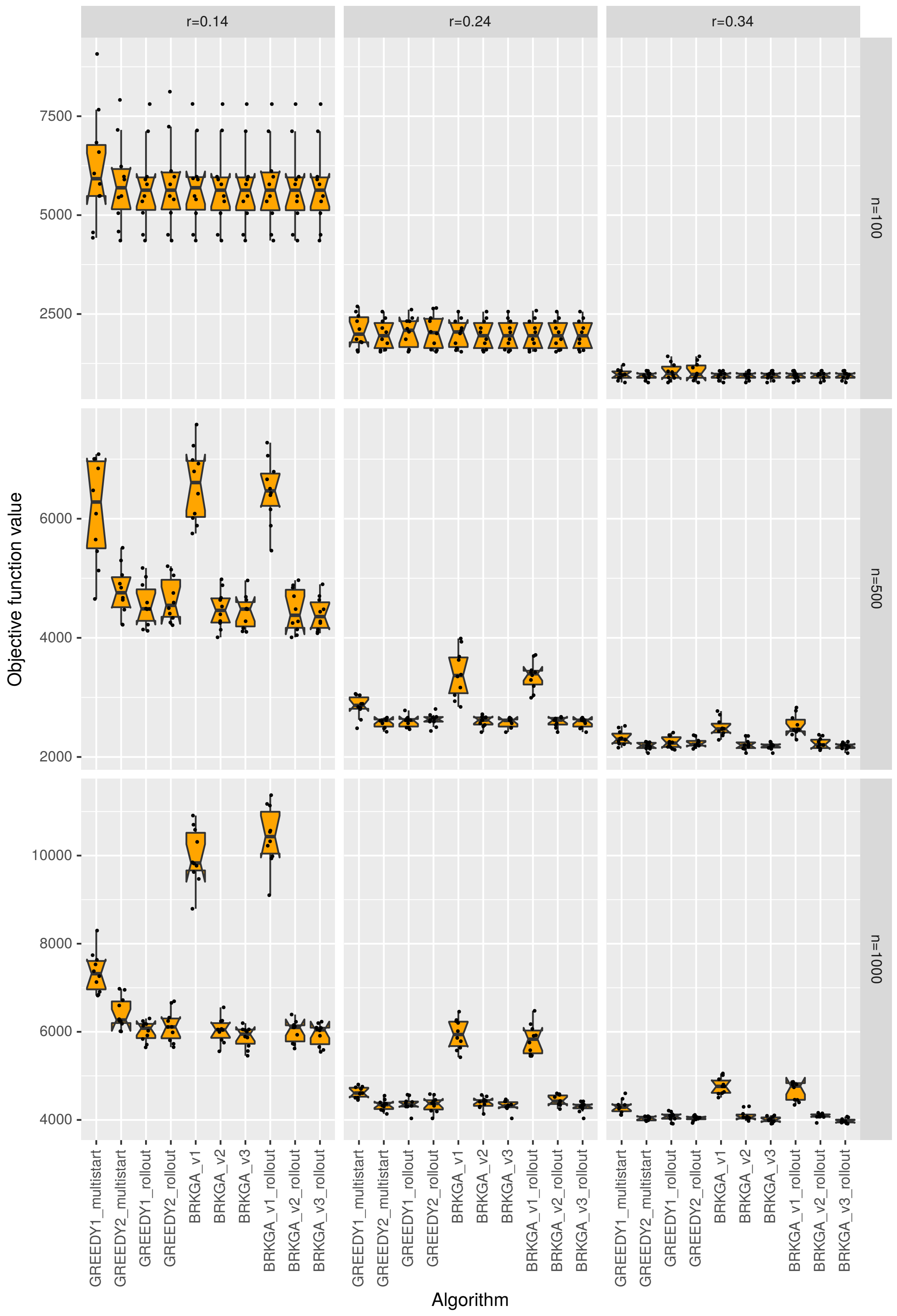
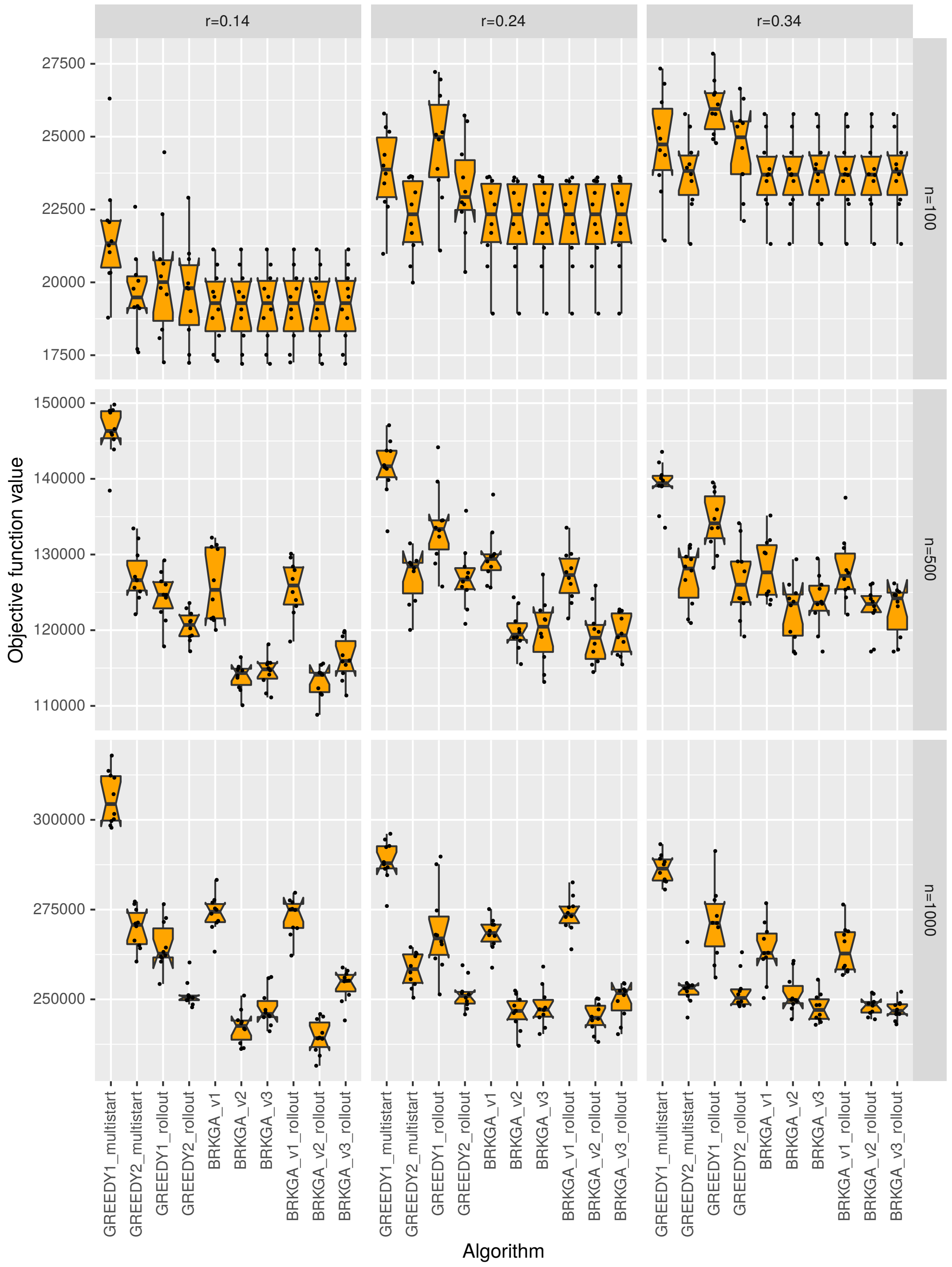


Figure 4.6: Notched boxplots for random geometric graphs with an edge-oriented weight scheme.



- In general, the results obtained from our developed greedy algorithms are not far from the ones achieved by the genetic algorithms, specially in graphs with a neutral or vertex-oriented weight generation scheme (see Figures 4.2 and 4.5).

Tables 4.11 and 4.12 show the results obtained by all Brkga versions. An extra column provides the best result found by the existing algorithms from the literature, in addition to the information by which algorithm this result was achieved. Furthermore, inequality and equality symbols are used to indicate if our approaches performed better, equal or worse than this best result from the literature. The results allow us to draw the following conclusions:

- Some minor differences are observed regarding the behaviour of our algorithms in random and random geometric graphs. In the latter ones, the obtained results seem to be closer to the best-known result than in the former ones.

- In random graphs, Brkga-V2 and Brkga-V2-rollout are, in general, the best-performing algorithms of the ones proposed in this thesis, followed by Brkga-V3 with its respective rollout adaptation. In random geometric graphs, Brkga-V2-rollout and Brkga-V3-rollout outperform the other approaches. They are followed by their respective standard versions, especially in smaller graphs.

- The third decoder seems to outperform the others for graphs in which the vertices have higher weights than the edges (see Figure 4.2). However, the results obtained in graphs generated with an edge-oriented scheme are worse compared to the ones achieved by the second decoder (see Figure 4.3). These differences are not as clear in geometric graphs.

- Version 1 of the Brkga algorithms is only competitive in the context of small instances (specially in random graphs). However, its performance decays as the number of nodes increases. This difference is emphasized in node-oriented graphs (see Figures 4.2 and 4.5)

- Nothing significant can be said regarding computation time, as a clear trend is not perceived.

- In general, the Brkga algorithms perform similarly to Ilp and to Pbig (including the Cmsa version) in graphs of smaller order.

- Concerning larger instances, our proposed algorithms do not perform as well as Pbig-Cmsa, which is the best-performing algorithm from the literature. In graphs of order 500, the results obtained by Brkga-V2-Rollout and Brkga-V3-Rollout are comparable to the ones from Pbig, and even superior in some cases.

Chapter 5

Final words

5.1 Conclusions

In this thesis, we have developed and implemented ten different algorithmic approaches for solving the WID problem using different techniques. A deep experimental evaluation was performed in order to test and compare our proposals to the ones from the literature.

On the one hand, four new greedy-based approaches were developed using two deterministic greedy heuristics from the literature. A rollout mechanism was applied to both. Additionally, we developed a generator of probabilistic greedy solutions. The experimental results show that these modifications far exceed the performance of the original heuristics.

On the other hand, we proposed three different decoders for a BRKGA. This element translates any vector of random numbers into valid solutions to the problem. The first two proceed by generating a permutation of vertices, while the third one uses the values of the individual to bias the original greedy function weights. Furthermore, a rollout mechanism was also applied to each of these decoders. The obtained results conclude that the BRKGAs are competitive to the state-of-art algorithm in small order graphs.

Although our approaches could not outperform the state of the art in larger graphs, we hope that the observations from this work will help future studies on this problem.

5.2 Future work

Due to the limited time available to carry out this thesis, we could not implement many of the ideas we had during its development.

The most important pending work, undoubtedly, is to carry out a more fine-grained tuning. As we explained in the previous section, the set of available instances was split into three subsets (concerning the graph sizes) for which the parameters of the algorithms were then adjusted. Nonetheless, the approaches from the literature were tuned in a more fine-grained way. In particular, the available instances were split into nine different subsets, one for each combination of graph type, number of nodes and weight generation scheme. For this reason, we think that the comparison was not totally fair, as a finer tuning would surely improve the performance of our algorithms in certain cases. Although we would have liked to carry out the same process, it was impossible due to the lack of time, as our simpler tuning process already took too long (around three weeks).

In the following subsections, we introduce some thoughts that could be explored in future works.

5.2.1 Parallel computing

Parallelism has long been employed in high-performance computing. Lately, it has gained interest due to the physical constraints preventing frequency scaling. In this subsection, we present some ideas for improving the performance of our algorithms by using two different hardware approaches.

CPU

The first option is to make use of multi-core systems and architectures. On the one hand, genetic algorithms have a natural parallel implementation. One way to proceed is by parallelizing certain genetic operations, while the other is by using multiple populations that evolve independently.

1. Parallelization of genetic operators
   Some of the operations performed during the evolution of a population can be computed in parallel, since they perform independent operations. In a BRKGA, these involve:

   - Generating the initial population (of size p).
   - Generating p_m mutants in each iteration.
   - Crossover (combining parents to produce offspring).
   - Fitness evaluation (decoder).

   The first three operators are not as computationally expensive as the fourth, so their parallelization would not produce a noticeable improvement. However, the decoding and evaluation of an individual accounts for most of the overall execution time. Thus, a significant speedup could be expected.

2. Multiple populations
   Another way to take advantage of having multiple processors is by having more than one population at once. Although this is also possible in sequential environments (pseudo-parallel execution), this implementation shines brighter when they are processed concurrently. This approach usually delivers better results when compared to single-population algorithms, finding the global optimum more frequently.

   In this model, each population evolves independently from the others. At the end, the individual with the best fitness value is selected as the proposed solution. This design is often called *island model*, because it resembles real-life situations where individuals of a kind may evolve separately by living isolated. Therefore, a population is also called an *island*.

As these multiple populations do not depend on each other, they can all run in parallel in various cores. Note that individuals from different populations can differ remarkably, as they have divergent mutation and crossover history. These differences can be even more accentuated by using distinct genetic operators in each island. For example, one could change the way an offspring is created, the allele inheritance probability or even the population size. This diversity helps when exploring the search space, making it harder to get stuck in local minima. To maintain a uniform evaluation criterion for every individual, the same fitness function must be used.

Although this independent evolution has some advantages, it might also be beneficial that some individuals from an island could move to others. This would allow fitter individuals to prevent and assist other populations getting permanently stuck in local minima by improving the quality of their individuals. For this purpose, these models usually let the populations evolve independently and, after a certain number of generations (called *isolation time*), a certain number of individuals is passed from each population to the others.

The genetic operator in charge of exchanging solutions is called *migration*. To determine the level of genetic diversity, the following variables are used:

- Migration rate: Determines the number of individuals distributed.
- Selection method: Determines which individuals from an island are chosen to migrate. Some examples are: selecting the fittest of the population or choosing some in a random manner.
- Topology: Determines among which populations the exchange is produced. For example, the migration could take place between all populations (unresticted) or only in a pre-defined neighbourhood.
- Replacement policy: Determines which individuals are substituted in the destination population. As islands usually have fixed sizes, they must make room for the newcomers. The most usual ways to proceed are by replacing the worse individuals or some at random.

It is important to point out that having, for example, a too permissive migration policy could relax the property of independent evolution, causing the populations to lose their diversity and converge to the same area in the search space, which is not desired.

It is clear to see that the second option is far better than the first, as it allows for more profitable alternatives. The multi-population approach, though, requires a framework that needs to be adjusted, thus it was not possible for us to implement this framework in the limited time of this project. However, this second alternative could be applied to the best-performing BRKGAs in order to improve their performance in the future.

On the other hand, parallelization techniques could also be applied to our greedy approaches. A reduction in the execution time of the ROLLOUT versions could be achieved by concurrently calculating the projections of each possible vertex at each iteration. Concerning the MULTISTART versions, we could have multiple solution generators running in parallel, returning the best solution found in general. Furthermore, each one could have distinct parameter settings in order to work with different degrees of greediness.

One simple way to include parallelization in our code is by using `OpenMP`, an API[1] that supports multi-platform shared memory multiprocessing programming. Another option could be an hybrid approach with both `OpenMP` and `MPI` (Message Passing Interface) for a better parallelism in computer clusters.

GPU

As previously mentioned, better performances in CPUs cannot be achieved by increasing their frequency. While consumption and heat dissipation deeply influences the design of new processors, the usage of GPUs (Graphic Processing Units) has grown in popularity over the last years.

Both units have completely opposed design philosophies. On the one hand, CPUs are aimed to reduce latency, making use of large caches, branch predictors and powerful ALUs[2]. On the other hand, the goal of GPUs is to increase the throughput. Despite having small caches and simple control units, they make use of a massive number of threads to mask their latency. GPUs are, thus, parallel by nature.

The main idea, then, is to adapt our BRKGA to exploit this power. Nowadays, the best option is to use `CUDA` (Compute Unified Device Architecture), the most popular framework (property of nVidia) designed for this purpose.

CUDA makes use of the following hardware model of a graphic card. The GPU is divided into various SIMD[3] multiprocessors. Accessing the main device memory is very slow, but these multiprocessors contain fast shared memory to overcome this weakness.

This hardware is mapped to the CUDA software model. There, one thread corresponds to a processor. A set of threads makes what is called a block (multiprocessor). These threads within a block share some memory, and can be easily synchronized.

After looking closely at it, it is easy to notice the similarities between this model and

[1]Application Programming Interface

[2]Arithmetic Logic Unit, an electronic circuit that performs arithmetic and bitwise operations on binary numbers.

[3]Single Instruction, Multiple Data. This refers to an instruction set available mostly on all processors, where the same operation is performed on multiple data points simultaneously, as opposed to executing multiple instructions.

a GA. Threads could be seen as individuals inside a package (block) or population, and these blocks could represent different islands. This is, then, an abstraction of the previously mentioned island model.

The algorithm could be implemented as follows. To begin with, each block would map to a population, which could be stored in shared memory for a better performance. Furthermore, each thread inside a block would take care of one individual and its associated calculations (randomly generating it, creating its offsprings, etc.). Therefore, all the necessary operations in a generation would be done concurrently. The model and mapping can be visualized in Figure 5.1.

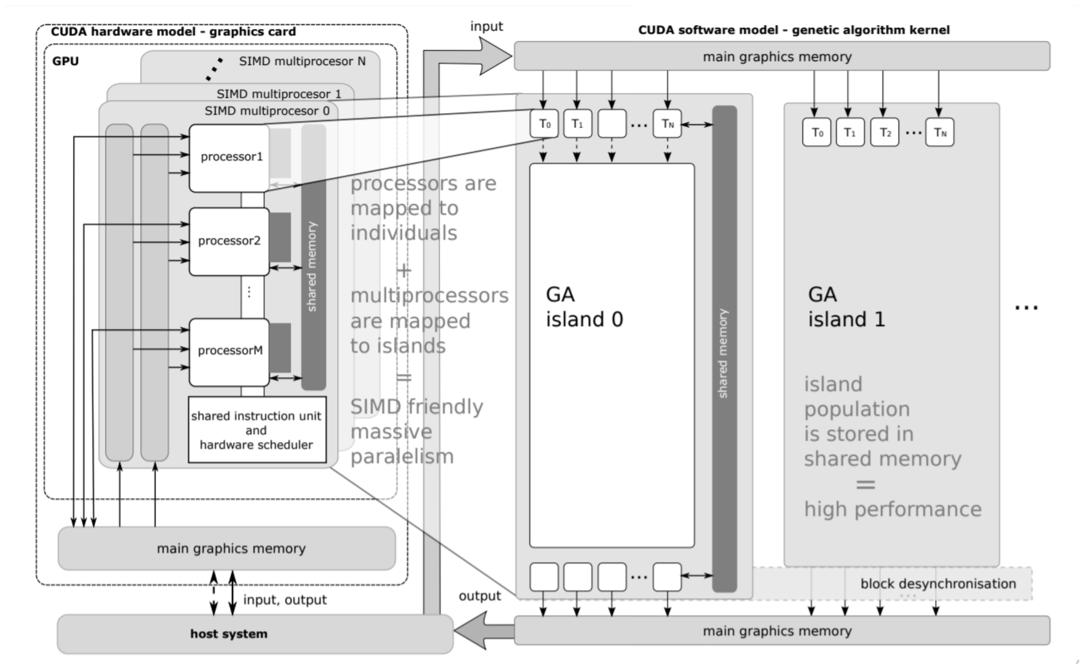

CUDA hardware model - graphics card
GPU
SIMD multiprocesor N
SIMD multiprocesor 1
SIMD multiprocesor 0
processor1
processor2
processorM
shared memory
shared instruction unit and hardware scheduler
main graphics memory
input, output
host system
input
output
CUDA software model - genetic algorithm kernel
main graphics memory
processors are mapped to individuals + multiprocessors are mapped to islands = SIMD friendly massive paralelism
GA island 0
GA island 1
island population is stored in shared memory = high performance
block desynchronisation
main graphics memory

Figure 5.1: Mapping CUDA hardware model to software model. Extracted from [24].

Nevertheless, our BRKGA has some characteristics that are not optimal for this parallelism: load unbalance, idle threads, etc., the algorithm could be adapted to this purpose. However, the need of constantly accessing the data structures where the graph information is stored (in order to calculate the individuals' fitness) would be a drawback. As it is impossible to store the data in shared memory (due to its size) this results in additional latency for accessing the main memory.

5.2.2 Generating greedy individuals

After observing the advantages of generating probabilistic greedy solutions, we thought about incorporating them as individuals in our BRKGAs. This injection of solutions could guide the algorithm through better areas in the search space than randomly generated ones, especially in those instances where the solutions provided by the greedy algorithms are close to the best ones obtained.

To achieve that, we have to translate solutions from the solution space to the hypercube, where the random-key individuals reside. Note that this is the exact opposite process of that performed by the decoder. We will only explain how to proceed for versions 1 and 2 of the algorithm, as version 3 is already influenced by the greedy heuristics.

To generate an individual π, we start by obtaining a solution $S \subseteq V$ using one of the multistart greedy versions. Note that $V \setminus S$ (the relative complement of S in V) represents the set of non-selected vertices, here denoted by $\not{S}$. Once completed, sets S and $\not{S}$ are shuffled independently, and a permutation P of vertices is created by concatenating S and $\not{S}$ Now, we have a permutation of vertices where the selected nodes appear before the non-selected. Note that shuffling the previous sets does not affect the solution itself, and it is done to avoid an undesired pattern. Furthermore, a vector R of size $|V|$ with uniformly distributed random values in range [0,1] is generated. The vector is then sorted in increasing order. Once we have all these elements, we proceed in a different way depending on which decoder (version) for the BRKGA is used.

Version 1

In this version, a vertex v in the permutation is translated to the random-key individual by taking the value in the vth position of the random vector. More formally, if v is the vertex in position i of permutation P ($v := p_i$), then $\pi_i := r_v$. See Figure 5.2 for an example.

Version 2

Here, if v is the vertex in position i of permutation P (p_i), then $\pi_v := r_{|V|-i}$ (that corresponds to the ith bigger value in the random vector R). See Figure 5.3 for an example.

5.2.3 Other improvements

After observing the behaviour of our proposed algorithms, we came up with some ideas that could be of interest, with a special focus on the BRKGA.

While testing the algorithm we noticed that, occasionally, the elite population got filled with identical or extremely similar individuals after some generations. This phenomenon is likely to happen, as there is no control over repeating individuals. This problem causes the algorithm to lose diversity, as elite individuals are preserved and influence offspring creation during forthcoming generations. One possible solution could be to adjust the value of some of the parameters during execution time. For example, if the algorithm detects that the number of identical or quasi-identical individuals surpass a given threshold, it could reduce the size of the elite population or decrease the allele inheritance probability so the offspring individuals are more likely to obtain genetic information from mutants.

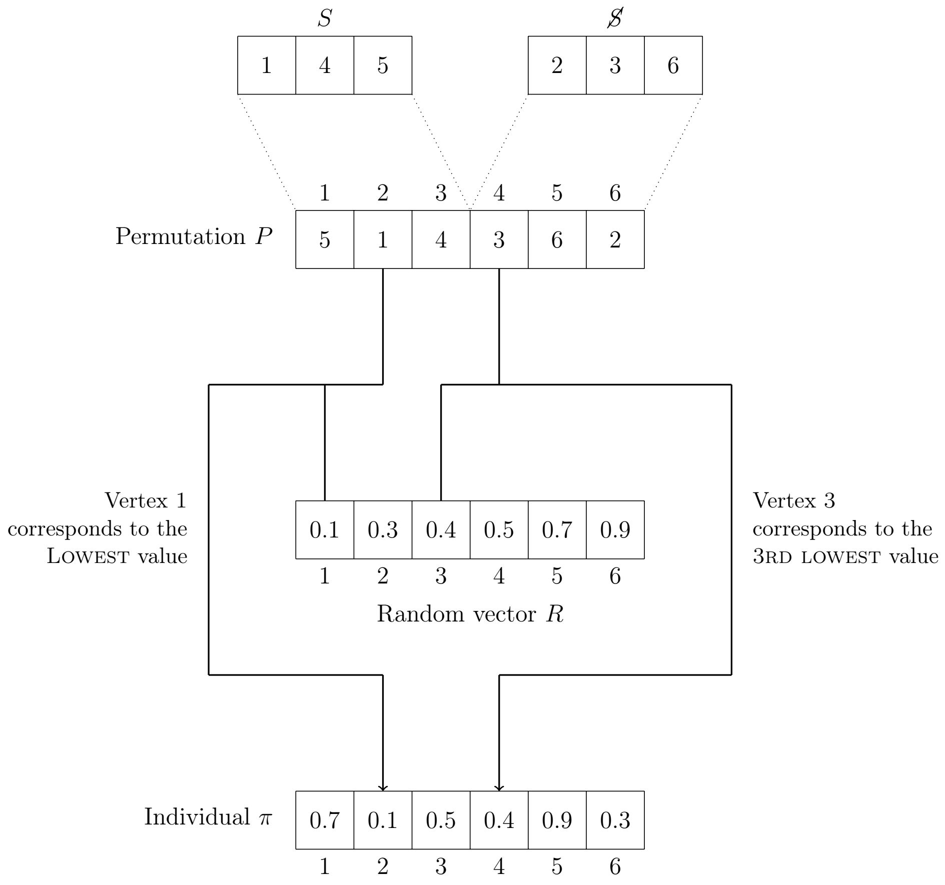


Figure 5.2: Example: transformation from solution space to the hypercube (version 1).

This could also be applied when the algorithm gets stuck during certain generations in a local minimum.

Another interesting point is related to the behaviour of the third decoder and the influence of both greedy heuristics. Recall that this version takes the best value obtained by one of the two biased versions of the greedy algorithms. It could happen that, in some instances, one greedy heuristic dominates the other one, meaning that, most individuals are better evaluated by this greedy heuristic. Thus, the computation time wasted calculating the value of the other heuristic could be saved. One example could be that, in graphs with a vertex-oriented weight scheme, the evaluation with GREEDY1 resulted superior.

Two ways of controlling this aspect came to our mind. One would be the use of only one greedy depending on the characteristics of the instances to which the algorithm is applied. Therefore, the programmer itself would select the greedy version beforehand. The other is to let the algorithm learn which heuristic results better during the algorithms' execution. This branch of EA are called Co-Evolutionary Algorithms (CoEAs), whose main idea is having more than one evaluation criteria and changing them dynamically according to environmental changes [18].

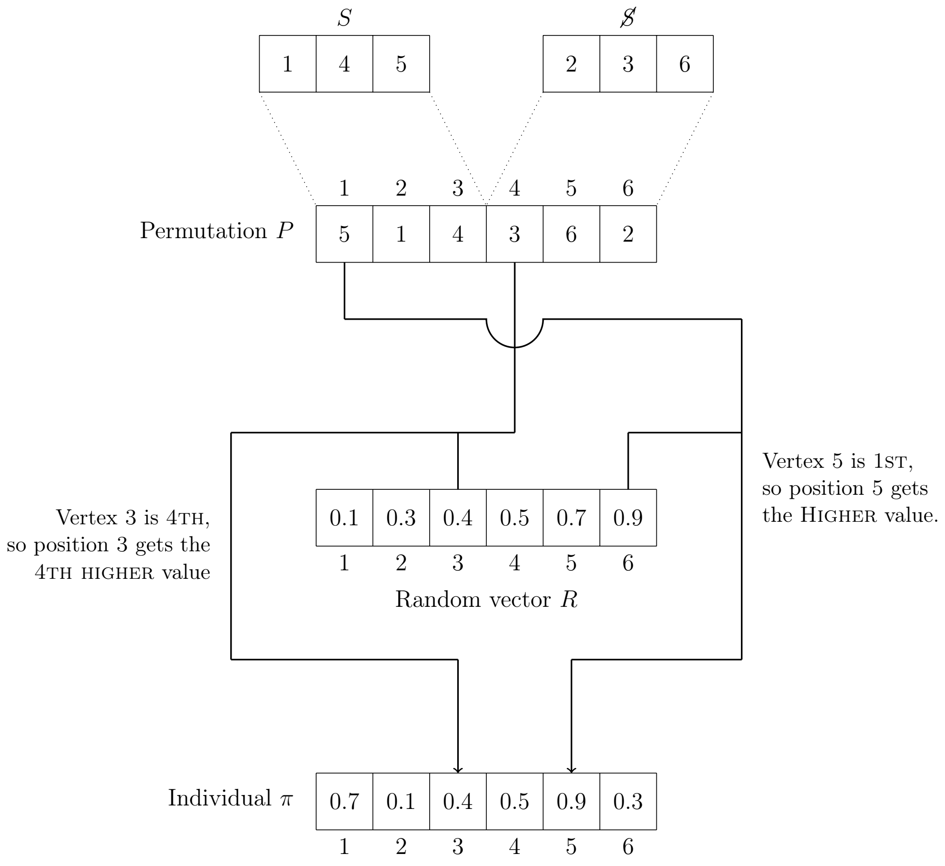


Figure 5.3: Example: transformation from solution space to the hypercube (version 2).

The last improvement could be the application of the BRKGA within the Constuct, Merge, Solve & Adapt (CMSA) framework, which was introduced in [5]. This framework divides tackled problem instances into smaller sub-instances and then solves them by applying a metaheuristic or other algorithmic approaches, taking profit of the fact that moderate size instances can be solved rather fast to optimality (even by using exact solvers). This higher-level framework has been proved to improve the performance of many approaches. In fact, the CMSA was used in [23], where it achieved better results than the standalone version of the PBIG algorithm. Therefore, we think that the application of our genetic algorithm within this framework would enhance its performance. This is especially because results showed that this approach performs well in instances of small size.

5.3 Technical competences

At the beginning of the project, we established, in agreement with the thesis director, some technical competences to comply with. Now that the work has come to an end, we report if these requirements have been accomplished.

- **CCO1.1:** *To evaluate the computational complexity of a problem, know about algorithmic strategies which can lead to its resolution and recommend, develop and implement the one that guarantees the best performance according to the established requirements.* [In depth]

  The goal of this thesis completely coincides with the description of this technical competence. Our approaches were designed to comply with the time requirements while trying to achieve good quality solutions to the tackled problem. Hence, its achievement is justified.

- **CCO3.1:** *To implement critical code following execution time, efficiency and security criteria.* [Enough]

  As previously stated, approximate algorithms are a good balance between complete (exact) results and a reasonable execution time, being a more appropriate option in large-size problems where exact algorithms take too long to compute. Concerning the implementation of our algorithms, we focused on being as efficient as possible, using suitable data structures for this purpose.

- **CCO3.2:** *To program considering the hardware architecture, both in assembly and high-level.* [A little bit]

  Even though our algorithms are designed to work on any architecture, we have considered the possibility of optimizing them by parallelizing certain areas of the code in order to take profit of architectures with multiple processors or the power of graphic cards. These ideas were already discussed in Section 5.2.1.

Appendix A

Project management

This appendix contains all the work done during the Project Management course, including the initial planing and estimated cost of the project and the comparison with the final ones. Note that this was carried out at the beginning of the project, so a future tense is used. However, sections contrasting estimations with definitive versions are written in present.

A.1 Project planning

In this section, we aim to describe the tasks in which the project will be divided in, specifying the action plan we will follow in order to finish the project in the desired time.

Being this a research project, and due to the use of agile methodologies, the initial planning is subject to modifications and could be revised and updated during the course of the thesis.

A.1.1 Schedule

The estimated duration of this project is of 5 months, from the beginning of February until July, where the final thesis presentations take place.

A.1.2 Methodology

As the development time of the project is relatively tight, an agile methodology seems the best way to approach it. Weekly meetings with the director and co-director will be held, offering a vision of the current state of the project and whether it is found within the expected planning or not. In addition, constant feedback from them will be received to solve the problems as briefly as possible.

The nature of our project involves constant changes to its objective and requirements. This factor leads us to the use of an agile methodology, such as Extreme Programming (XP). Extreme programming is intended to improve responsiveness to changing customer requirements, using short development cycles and introducing checkpoints at which the new requirements can be adopted.

In order to follow this schema, we will divide our planning in cycles of about 1 or 2 weeks. In each cycle, we will begin by defining the objectives and tasks to be achieved, changing the previously fixed requirements if needed.

A.1.3 Task description

Acquiring background

As in any research project, the first thing to do is gathering information about the subject of study. Therefore, before starting the proper project development, I read some documentation about the topic (such as previous papers tackling this problem or relevant related work) in order to get a better understanding of the field of study.

In addition, I checked the code of the previously developed algorithms in [23] so I could have a clear idea of the starting point of our project.

- Estimated Time
  The process took around two weeks.

- Resources
  This task mainly required human resources (to read and understand the documents).

Project planning

This task is completely covered by the GEP course. Here, we will make an overview of the project, planning and scheduling the work involved in order to complete the thesis in time. It is divided into the following sections:

1. Context and scope of the project: Defining the scope of the project, specifying the stakeholders (target audience, users and beneficiaries) and doing a deep literature review on the subject under study.

2. Project planning: Describing the main tasks to be completed and its order of execution, as well as looking for alternative solutions and studying potential deviations. Required resources will be specified.

3. Budget and sustainability. Analyzing the sustainability of the project on its three different dimensions and estimating the project costs and budget.

- Estimated Time
  The estimated time for each section is one week. This task will end with an oral presentation of the delivered reports.

- Resources

1. Human resources: A project manager, responsible for the project roadmap and documentation.
2. Material resources: A personal computer and a word processor, such as Microsoft Word, Google Docs or LaTeX.

Implementation of the algorithms

In this phase, we will develop new algorithms to solve the problem, with special focus on tackling large graphs. It can be divided into three main stages:

1. Research: Finding and studying documentation about the kind of algorithm to be implemented.
2. Design and implementation: Constructing and coding the selected algorithm and/or its modifications.
3. Test: Checking the correct behaviour of the proposed solution.

Note that this phase will be cyclic, as we will be repeating it for each algorithm or modification we implement. In addition, a short documentation of the performed work will be written at the end of each cycle.

Our initial plan is to apply a rollout mechanism [4](a form of sequential optimization) to the greedy algorithms already implemented in the article we base our work in (which will take us around two or three weeks).

The next step will be the implementation of a Biased Random-Key Genetic Algorithm [15], focusing on building the problem specific decoder. This sub-task will take us about one and a half month, as potential improvements will be sought.

- Estimated Time
  Around two months.
- Resources
  1. Human resources: A software developer (responsible for implementing the algorithms) and a project manager (responsible for controlling the guidelines).
  2. Material resources: A personal computer, an IDE (such as Cloud9) and a file hosting service (Dropbox).

Experimentation and results

In this task, a deep experimental evaluation will be performed, in order to check the correctness and performance of the developed algorithms. Each one will be tested with graphs of different types, sizes and densities, to get a real numerical comparison about their efficiency and effectiveness. This will also help us to monitor the evolution of the project and possible deviations.

- Estimated Time
  The approximate duration of this task will be of one month, but may vary depending on the number of algorithms finally implemented and its execution time.

- Resources

  1. Human resources: A person in charge of testing the required algorithms and recording the results.
  2. Material resources: A computer cluster (such as the High Performance Cluster (HPC) at the UPC), where we can send out tests to execute (as, otherwise, it would take an excessive amount of hours).

Documentation

This will be the last stage of the project, where the final thesis report will be written. It will include an explanation of the context of study, the implementation of our algorithms and the evaluation of the obtained results. This task (and the entire project) will end with an oral defence of the thesis in a public session.

- Estimated Time
  Around one month.

- Resources

  1. Human resources: A project manager, responsible for writing the required report.
  2. Material resources: A personal computer and a word processor, such as Microsoft Word or LaTeX.

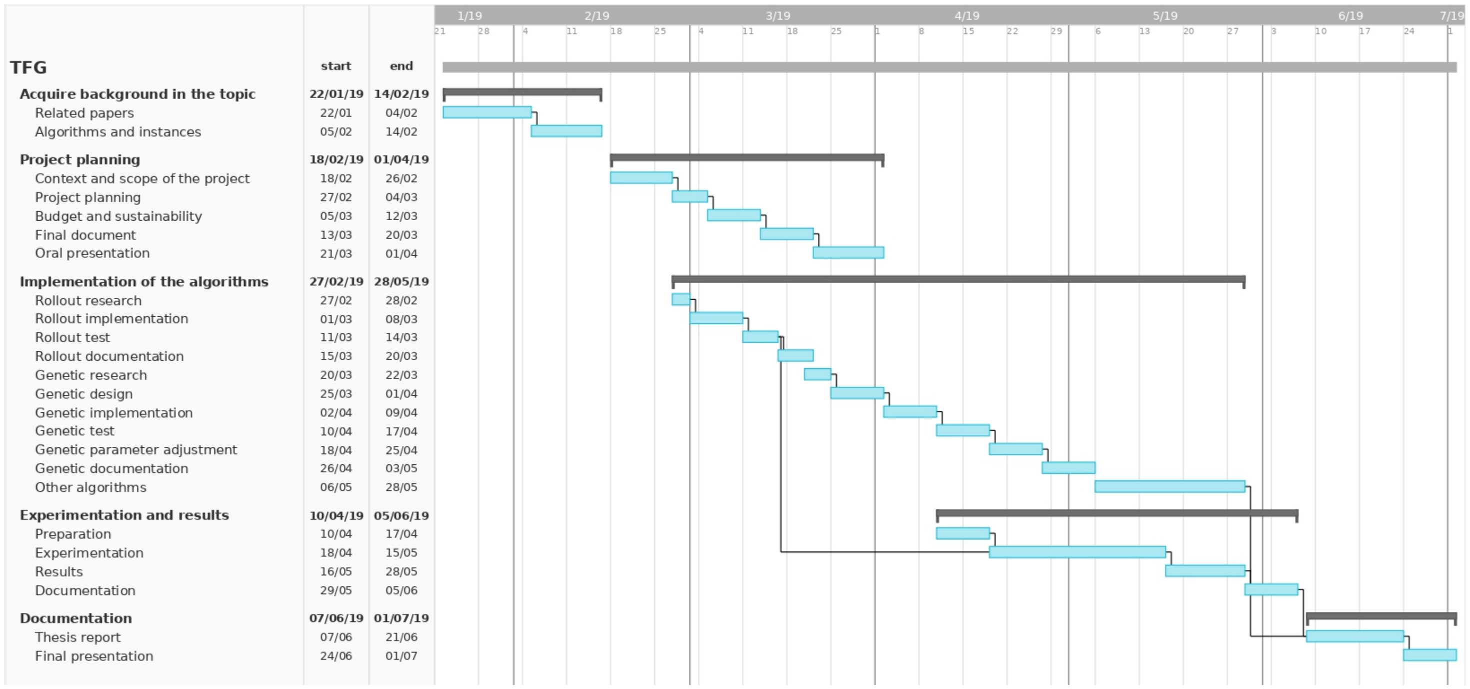
A.1.4 Estimated Time

Table A.1 summarizes the estimated number of hours dedicated to each task.

A.1.5 Final planning

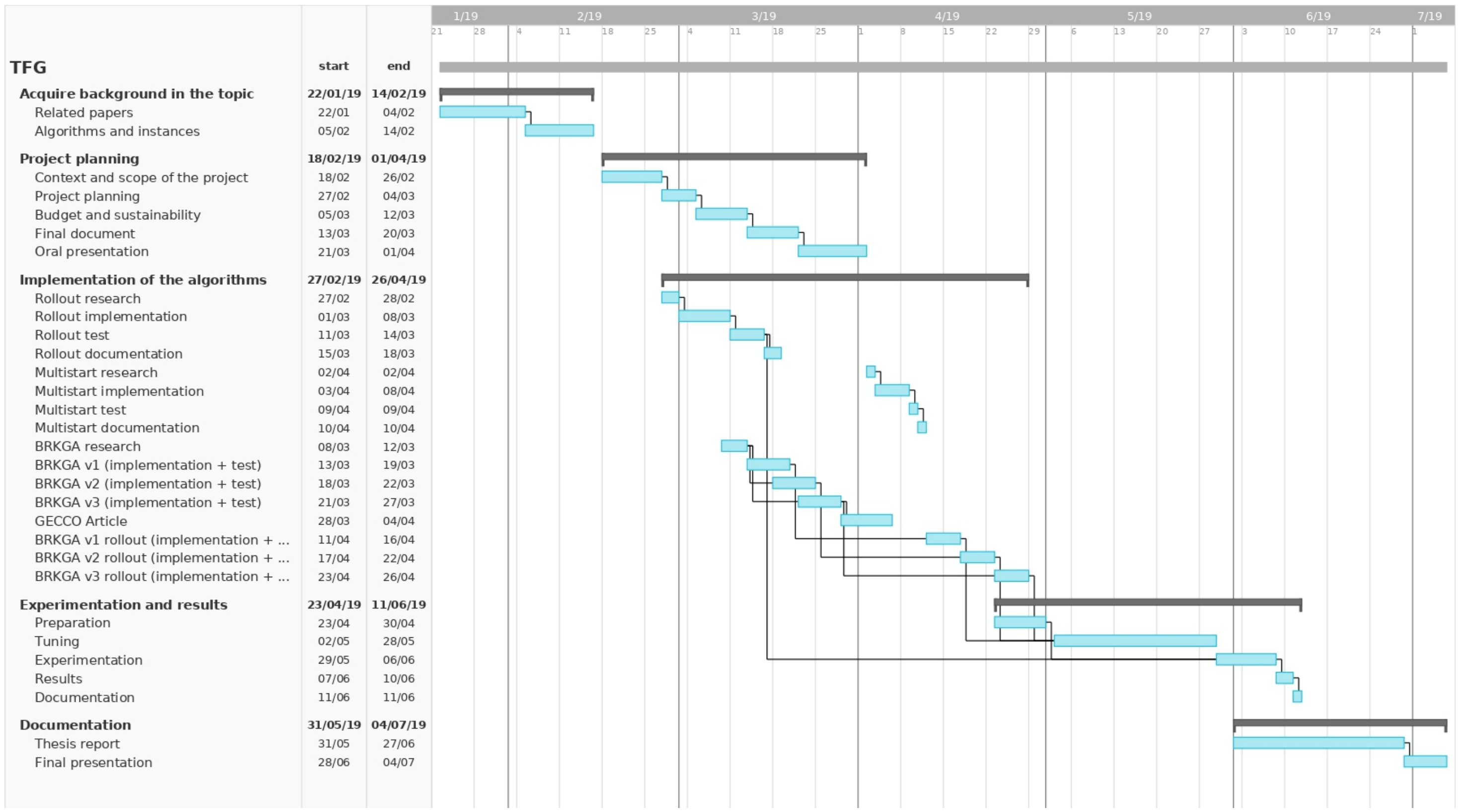
Now that the project has come to an end, we can compare the initial task planning (presented as a Gantt Chart in Figure A.1) and the final version (Figure A.2) in order to make an analysis of the changes.

As we can observe, both plannings are quite similar. At the beginning of the project we did not know which algorithms were going to be implemented, so we left room for potential changes in the third section. At the end, though, the implementation time was shorter than expected, as the project got on track once we started designing the genetic algorithm. The tuning phase was moved to the experimental section and lasted longer than expected (more than three weeks). However, the quickness of the experiments compensated that deviation, so the project has been finished within the expected time.



TFG	start	end
Acquire background in the topic	**22/01/19**	**14/02/19**
Related papers	22/01	04/02
Algorithms and instances	05/02	14/02
Project planning	**18/02/19**	**01/04/19**
Context and scope of the project	18/02	26/02
Project planning	27/02	04/03
Budget and sustainability	05/03	12/03
Final document	13/03	20/03
Oral presentation	21/03	01/04
Implementation of the algorithms	**27/02/19**	**28/05/19**
Rollout research	27/02	28/02
Rollout implementation	01/03	08/03
Rollout test	11/03	14/03
Rollout documentation	15/03	20/03
Genetic research	20/03	22/03
Genetic design	25/03	01/04
Genetic implementation	02/04	09/04
Genetic test	10/04	17/04
Genetic parameter adjustment	18/04	25/04
Genetic documentation	26/04	03/05
Other algorithms	06/05	28/05
Experimentation and results	**10/04/19**	**05/06/19**
Preparation	10/04	17/04
Experimentation	18/04	15/05
Results	16/05	28/05
Documentation	29/05	05/06
Documentation	**07/06/19**	**01/07/19**
Thesis report	07/06	21/06
Final presentation	24/06	01/07

Figure A.1: Initial planning. Gantt chart.



TFG	start	end
Acquire background in the topic	**22/01/19**	**14/02/19**
Related papers	22/01	04/02
Algorithms and instances	05/02	14/02
Project planning	**18/02/19**	**01/04/19**
Context and scope of the project	18/02	26/02
Project planning	27/02	04/03
Budget and sustainability	05/03	12/03
Final document	13/03	20/03
Oral presentation	21/03	01/04
Implementation of the algorithms	**27/02/19**	**26/04/19**
Rollout research	27/02	28/02
Rollout implementation	01/03	08/03
Rollout test	11/03	14/03
Rollout documentation	15/03	18/03
Multistart research	02/04	02/04
Multistart implementation	03/04	08/04
Multistart test	09/04	09/04
Multistart documentation	10/04	10/04
BRKGA research	08/03	12/03
BRKGA v1 (implementation + test)	13/03	19/03
BRKGA v2 (implementation + test)	18/03	22/03
BRKGA v3 (implementation + test)	21/03	27/03
GECCO Article	28/03	04/04
BRKGA v1 rollout (implementation + ...	11/04	16/04
BRKGA v2 rollout (implementation + ...	17/04	22/04
BRKGA v3 rollout (implementation + ...	23/04	26/04
Experimentation and results	**23/04/19**	**11/06/19**
Preparation	23/04	30/04
Tuning	02/05	28/05
Experimentation	29/05	06/06
Results	07/06	10/06
Documentation	11/06	11/06
Documentation	**31/05/19**	**04/07/19**
Thesis report	31/05	27/06
Final presentation	28/06	04/07

Figure A.2: Final planning. Gantt chart.

Task	**Estimated duration (h)**
Acquiring background	20
Project planning	90
Implementation of the algorithms	200
Experimentation and results	100
Documentation	40
Total	450

Table A.1: Summary of the estimated time spent on each task

A.2 Budget

In the following section, we aim to make an estimation of the total budget needed to carry out our project. Costs will be calculated taking into account the project planning (Gantt chart) and resources stated in the previous section.

For the budget, we will consider human, hardware and software resources. In addition, indirect and unexpected costs will be contemplated. To conclude, an analysis of potential deviations will be made in order to know how they can affect our initial calculation.

This budget is subject to changes depending on the project development. A comparison between this initial estimation and the real costs is explained in Section A.2.7.

A.2.1 Human resources

Although this project will be developed by one person, he will take three different roles: project manager, software developer and tester. The following table provides an estimation of the human resources costs, taking into account the tasks and hours assigned to each role, tasking as a reference the mean salary received for this kind of jobs in I&T companies. Table A.2 contains the cost for the human resources.

Role	**Hours(h)**	**Price per hour (€/h)**	**Cost (€)**
Project manager	90	50	4500
Software developer	230	30	6900
Tester	130	30	3900
Total	450	-	15300

Table A.2: Human resources budget.

Considering this is a kind of project usually done by a research fellow from a university, costs could be adapted to the gross salary they receive. In addition, this project could also be developed by a student granted a research scholarship (with an estimated 7€/h salary), considerably reducing the costs for human resources.

A.2.2 Hardware resources

In order to carry out our project, we will need a workstation where we can develop the algorithms and write the reports. A laptop will be used for this purpose, as they are more energy-efficient than a desktop computer and will allow us to work in different places. Although a computer cluster will be needed, its cost will not be included here, as the one we will be using (the High-Performance Cluster at the UPC) belongs to the Computer Science Department (and maintained by the RDlab).

Table A.3 contains the cost of the hardware that we are going to use in the project.

Product	**Price(€)**	**Units**	**Lifespan**	**Amortization (€)**
Laptop	950[1]	1	5 years	80
Total	950	-	-	80

Table A.3: Hardware resources budget.

A.2.3 Software resources

In this project, we will be using the following software:

1. Cloud9, an online integrated development environment.
2. LaTeX/ Google Docs, a word processor.
3. TeamGantt, an online Gantt chart software.
4. Dropbox, a file hosting service.

Note that they are completely free tools, so there is no cost associated to software resources. In addition, all programs used at the implementation phase (such as previously developed algorithms or packages) are of free use (under the GNU General Public License).

A.2.4 Direct costs

Table A.4 contains an estimation of the direct costs mentioned above, split for the different tasks:

Task	**Estimated cost (€)**
Acquiring background	610
Project planning	2800
Implementation of the algorithms	6030
Experimentation and results	3930
Documentation	2010
Total	15380

Table A.4: Direct costs.

A.2.5 Indirect costs

Table A.5 contains an estimation of indirect costs not included in any of the previous categories, such as electricity, Internet and office supplies (paper, etc.).

Product	**Price**	**Units**	**Cost (€)**
Electricity	0.12€/kWh	60kWh	17.76
Internet	50€/month	5 months	250
Office supplies	100€	-	100
Total	-	-	367.76

Table A.5: Indirect costs.

A.2.6 Total budget

By adding all the previously provided budgets, we get the total estimated for this project, as shown in table 6. A 5% of contingency has been added in order to cover unexpected expenses (mainly to handle an increase in the developer's working hours). We think this quantity is enough, as a high deviation from the initial planning is not expected.

A.2.7 Budget deviations

As the project has been completed in the estimated time and no major deviations from the initial planning have been detected, the cost of the project is within the stipulated budget. Although the computer cluster was free to use (as members of the university), we asked for an invoice to the RDlab in order to know the real cost of it if we were an external group not linked to the department.

Concept	**Estimated cost (€)**
Human resources	15300
Hardware resources	80
Software resources	0
Indirect costs	367.76
Subtotal	15747.76
Contingency(%)	787.4
Total	16535.16

Table A.6: Total budget.

The cluster has used a total of 66237943 seconds (18399.4 hours). Approximately, all of our processes employed 1 core and consumed almost 12Gbytes. A pack is equivalent to 60 minutes with 1 CPU core and 4Gbytes of RAM and costs 0.16€$+VAT$. We used 3 packs per hour (12Gbytes/4GBytes), so the total cost is $18399.4h \cdot 3packs \cdot 0.16$€$+VAT = 8831.71$€$+ VAT(21\%) = 10686.37$€. To this value, we have to add the RDlab support price during the 3 months with granted access to the cluster (20€$/month \cdot 3months + VAT = 72.6$€). The disk space cost was almost 0€, so it is not taken into account.

Appendix B

Sustainability report

B.1 Environmental dimension

It is clear to see that the main environmental impact of the project is the electricity consumption derived from the use of personal computers. In addition, computer clusters usually consume much energy, since they need to be well refrigerated.

In order to minimize the impact, energy consumption can be minimized by using the energy-saving mode, which reduces the computer's performance when possible.

Another major problem is that, for the obtainment of resources such as electricity or paper fabrication, processes which leave substantial ecological footprints on the environment are used. Although those issues go beyond our reach, we can contribute with small gestures, such as recycling or reusing paper and other materials.

On the other hand, being this an optimisation problem, it can be indirectly positive to the environment, as an improvement would enhance the performance of real-life applications modelled in terms of finding dominating sets in graphs.

B.2 Economic dimension

In previous sections, a detailed planning and estimated budget have been presented, including the resources needed to carry out the project and the quantified cost of the development phase.

We tried to use and adapt existing technologies for the implementation and testing of our proposed algorithms, reducing the cost of creating new ones.

The project was finished within the expected time. Therefore, no additional cost involving human resources has been considered. As we didn't create a product, there was no need to handle extra costs related to it, such as maintenance, commercialization or distribution.

B.3 Social dimension

Research projects of this kind do not have many social implications. Nevertheless, our proposed algorithms may serve as an example for other optimization problems, helping other researchers already investigating this problem (or similar ones). The use of open source software will facilitate this by giving access to the results to everyone interested.

B.4 Sustainability matrix

The sustainability matrix from the project is presented in Table B.1. The analysis is divided in three parts, corresponding to different stages or aspects of the project:

- Project put into production (PPP): Includes the planning, developing and implementation of the project.
- Exploitation: Once the project has been implemented until its dismantling.
- Risks: The eventualities that could impact more negatively than expected.

	Environmental	**Economic**	**Social**	**Range**
PPP	6	6	7	(0:10)
Exploitation	-2	8	9	(-10:10)
Risks	0	0	0	(-20:0)
Final evaluation		34		(-90:60)

Table B.1: Sustainability matrix

On the environmental analysis, the planning stage has not left a remarkable ecological footprint. The resources used where the necessary to carry out the thesis, and some actions to reduce the energy consumption have been considered. Regarding the experimental evaluation of our algorithms, the cluster has consumed more than expected, so a negative mark is assigned.

When talking about the economic dimension, it is important to note that the final cost of the project lies within the expected budget. During its exploitation, no additional costs are required.

Finally, on the social dimension, the project has signified a personal a professional growth. As mentioned in the previous section, the results and thoughts could be useful for forthcoming works.

Considering this thesis is essentially a research project (with no more possible negative implications), no risk has been associated to any of the three sustainability aspects.

Bibliography

[1] Diogo V. Andrade, Mauricio G. C. Resende, and Renato F. Werneck. Fast local search for the maximum independent set problem. *Journal of Heuristics*, 18(4):525–547, February 2012.

[2] T. Bäck, D.B. Fogel, and Z. Michalewicz. *Handbook of Evolutionary Computation*. Oxford University Press, New York, NY, 1997.

[3] James C. Bean. Genetic algorithms and random keys for sequencing and optimization. *ORSA Journal on Computing*, 6(2):154–160, May 1994.

[4] Dimitri P. Bertsekas. Rollout algorithms for discrete optimization: A survey. In *Handbook of Combinatorial Optimization*, pages 2989–3013. Springer New York, 2013.

[5] Christian Blum, Pedro Pinacho, Manuel López-Ibáñez, and José A. Lozano. Construct, merge, solve & adapt: A new general algorithm for combinatorial optimization. *Computers & Operations Research*, 68:75–88, 2016.

[6] Christian Blum, Jakob Puchinger, Günther R. Raidl, and Andrea Roli. Hybrid metaheuristics in combinatorial optimization: A survey. *Applied Soft Computing*, 11(6):4135–4151, 2011.

[7] Christian Blum and Günther R. Raidl. *Hybrid Metaheuristics*. Springer International Publishing, 2016.

[8] Salim Bouamama and Christian Blum. A hybrid algorithmic model for the minimum weight dominating set problem. *Simulation Modelling Practice and Theory*, 64:57–68, May 2016.

[9] G. J. Chang. The weighted independent domination problem is NP-complete for chordal graphs. *Discrete Applied Mathematics*, 143(1):351–352, 2004.

[10] Sachchida Nand Chaurasia and Alok Singh. A hybrid evolutionary algorithm with guided mutation for minimum weight dominating set. *Applied Intelligence*, 43(3):512–529, April 2015.

[11] M. Dorigo and T. Stützle. *Ant Colony Optimization*. MIT Press, Cambridge, MA, 2004.

[12] T. A. Feo and M. G. C. Resende. Greedy randomized adaptive search procedures. *Journal of Global Optimization*, 6:109–133, 1995.

[13] Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman, New York, 1979.

[14] F. Glover and M. Laguna. *Tabu Search*. Kluwer Academic Publishers, 1997.

[15] José Fernando Gonçalves and Mauricio G. C. Resende. Biased random-key genetic algorithms for combinatorial optimization. *Journal of Heuristics*, 17(5):487–525, August 2010.

[16] John H Holland. *Adaptation in natural and artificial systems: An introductory analysis with applications to biology, control, and artificial intelligence*. University of Michigan Press, June 1975.

[17] Yan Jin and Jin-Kao Hao. General swap-based multiple neighborhood tabu search for the maximum independent set problem. *Engineering Applications of Artificial Intelligence*, 37:20–33, January 2015.

[18] Jeniefer Kavetha. No . 0976-5697 coevolution evolutionary algorithm : A survey. 2013.

[19] S. Kirkpatrick, C. Gellat, and M. Vecchi. Optimization by simulated annealing. *Science*, 220:671–680, 1983.

[20] Manuel López-Ibáñez, Jérémie Dubois-Lacoste, Leslie Pérez Cáceres, Mauro Birattari, and Thomas Stützle. The irace package: Iterated racing for automatic algorithm configuration. *Operations Research Perspectives*, 3:43–58, 2016.

[21] H. R. Lourenço, O. Martin, and T. Stützle. Iterated local search. In F. Glover and G. Kochenberger, editors, *Handbook of Metaheuristics*, volume 57 of *International Series in Operations Research & Management Science*, pages 321–353. Kluwer Academic Publishers, Norwell, MA, 2002.

[22] Bruno Nogueira, Rian G. S. Pinheiro, and Anand Subramanian. A hybrid iterated local search heuristic for the maximum weight independent set problem. *Optimization Letters*, 12(3):567–583, March 2017.

[23] P. Pinacho Davidson, C. Blum, and J. A. Lozano. The weighted independent domination problem: Integer linear programming models and metaheuristic approaches. *European Journal of Operational Research*, 265(3):860–871, 2018.

[24] Petr Pospichal, Jiri Jaros, and Josef Schwarz. Parallel genetic algorithm on the CUDA architecture. In *Applications of Evolutionary Computation*, pages 442–451. Springer Berlin Heidelberg, 2010.

[25] Anupama Potluri and Alok Singh. Hybrid metaheuristic algorithms for minimum weight dominating set. *Applied Soft Computing*, 13(1):76–88, January 2013.

[26] Rubén Ruiz and Thomas Stützle. A simple and effective iterated greedy algorithm for the permutation flowshop scheduling problem. *European Journal of Operational Research*, 177(3):2033–2049, 2007.

[27] Chang Shun-Chieh, Liu Jia-Jie, and Wang Yue-Li. The weighted independent domination problem in series-parallel graphs. *Frontiers in Artificial Intelligence and Applications*, 274(Intelligent Systems and Applications):77–84, 2015.

[28] Yiyuan Wang, Shaowei Cai, and Minghao Yin. Local search for minimum weight dominating set with two-level configuration checking and frequency based scoring function. *Journal of Artificial Intelligence Research*, 58:267–295, February 2017.

[29] Yiyuan Wang, Jiejiang Chen, Huanyao Sun, and Minghao Yin. A memetic algorithm for minimum independent dominating set problem. *Neural Computing and Applications*, 30(8):2519–2529, January 2017.

[30] Yiyuan Wang, Ruizhi Li, Yupeng Zhou, and Minghao Yin. A path cost-based GRASP for minimum independent dominating set problem. *Neural Computing and Applications*, 28(S1):143–151, April 2016.