Masters' Degree in Informatics Engineering
Dissertation
Final report

Subset Selection Algorithms in Multiobjective Optimisation

Daniel Jorge Ramos Vaz
dvaz@student.dei.uc.pt

Adviser:
Luís Paquete

July 2, 2013

FCTUC DEPARTMENT
OF INFORMATICS ENGINEERING
FACULTY OF SCIENCES AND TECHNOLOGY
UNIVERSITY OF COIMBRA

Abstract

The main focus of this work is on the design and analysis of algorithms for finding a representative subset of solutions of a multiobjective combinatorial optimisation problem. This problem is recast as a particular optimisation problem.
Two types of problems are discussed in this report. The first problem consists in finding a subset of elements that is as close as possible to a reference set with respect to the ϵ-indicator. In particular, new results with respect to the correctness of a known algorithm and three new approaches that improve the current time complexity are presented and discussed. Their performance is also compared from an experimental point of view.
In the second problem, the goal is to find a subset of elements of a set according to some property of interest. Three properties are taken into account: Uniformity, Coverage, and ϵ-indicator. The multiobjective problem that arises from the combination of these properties is also discussed. Two approaches are presented for each problem: one that is based on principles of dynamic programming and other that solves a sequence of feasibility problems. The algorithms are assessed in terms of time complexity and running time.

Keywords

Algorithm design and analysis
Computational complexity
Multiobjective combinatorial optimization
Subset selection
ϵ-indicator

Resumo

O foco deste trabalho é o design e análise de algoritmos que encontrem um subconjunto representativo de soluções de um problema de optimização combinatória multiobjectivo. Este problema é reformulado como um determinado problema de optimização.
Dois tipos de problemas são discutidos neste documento. O primeiro problema consiste na pesquisa de um subconjunto de elementos que seja tão próximo quanto possível de um conjunto de referência, de acordo com o Indicador-ϵ. Concretamente, são apresentados e discutidos novos resultados respeitantes à exactidão de um algoritmo conhecido, bem como três novas abordagens que melhoram a sua complexidade temporal. O desempenho destas abordagens é ainda comparada de forma experimental.
No segundo problema, o objectivo é encontrar um subconjunto de elementos de um conjunto, de acordo com uma dada propriedade. Três propriedades diferentes são tidas em conta: Uniformidade, Cobertura e Indicador-ϵ. É ainda discutido o problema multiobjectivo resultante da combinação dessas propriedades. Duas abordagens são apresentadas para cada problema: uma delas é baseada em princípios de programação dinâmica, e a outra resolve vários problemas de exequibilidade em sequência. Os algoritmos são avaliados relativamente à complexidade temporal e ao tempo de execução.

Palavras-chave

Design e análise de algoritmos
Complexidade computacional
Optimização combinatória multiobjectivo
Selecção de subconjuntos
Indicador-ϵ

Acknowledgements

I would like to thank my family, particularly my parents and sister, for supporting me, providing a great familiar environment, and helping me get to where I am today. To my friends, thank you for the leisurely moments, which are part of a healthy lifestyle and coexist with the hard work. My thanks to my adviser, Prof. Luís Paquete, for guiding my work and for all the incentive and effort to make my thesis better. My thanks also go to the ECOS lab, for the company and work environment, and for allowing me to access and use their cluster. I would also like to thank Prof. Kathrin Klamroth and Prof. Michael Stiglmayr for providing different points of view and for the discussion of the results of the thesis. For the support and for the opportunity of visiting the University of Wuppertal, I also express my appreciation to the RepSys project – Representation systems with quality guarantees for multi-objective optimization problems, Germany/Portugal Bilateral Cooperation Research Project funded by DAAD/CRUP.

Contents

Chapter 1

Introduction

Optimisation problems arise in many real-life situations, such as finding a shortest path between two locations with a navigation system or finding a good scheduling of classes that maximises the preferences of the staff. In order to generate realistic solutions, there is the need to consider conflicting interaction between the players of the optimisation problem. For example, if we consider the preferences of the students when scheduling classes, these may be incompatible with those of the teachers: while teachers may want to have the classes concentrated in one or two days, students prefer to have classes spread over the week or only during the afternoon. Therefore, we may need to consider multiple objectives in order to solve more realistic problems.

When we have two or more conflicting objectives, it may be impossible to find a solution that satisfies all of them. One way to overcome this problem is to find a set of "trade-off" solutions, each of which representing a compromise between the objectives. Then, this set of solutions, which we denote by *trade-off set*, is presented to a decision maker that chooses the most appropriate solution for the problem at hand. Figure 1.1 presents several timetables as points, whose coordinates correspond to staff preferences (y-axis) and students' preferences (x-axis). If we choose the uppermost timetable, we are satisfying 38% of the staff preferences, but only 10% of the students' preferences. On the other hand, if we choose the third point from the left, we obtain a timetable that satisfies 27% of the staff preferences, while satisfying 31% of the students' preferences. This is a compromise solution, which does not completely satisfy both parties. However, we may also choose a timetable that favours students, like the rightmost point, which satisfies 54% of the students' preferences, while only 7% of the staff preferences.

Unfortunately, these problems may generate a large number of solutions, which may overload the decision maker. Therefore, it is important to find a small number of representative solutions. This describes the problem of *subset selection*, i.e. the problem of choosing a fixed number of elements from a given set according to some properties.

Since the representative subset can be seen as an approximation, we can use well-known measures of quality of approximations to characterise these sets. Two measures are commonly used: i) Uniformity, which is related to the distance between a solution and its neighbours and whose goal is to prevent large gaps in the chosen subset and ii) Coverage, which measures how close are the chosen elements from those that were not chosen [10]. Figure 1.2 presents two subsets of four solutions for the same timetable instance as shown in Figure 1.1. The subset represented on the left has high Uniformity, since the solutions are evenly spaced, i.e, the distance between neighbour solutions is

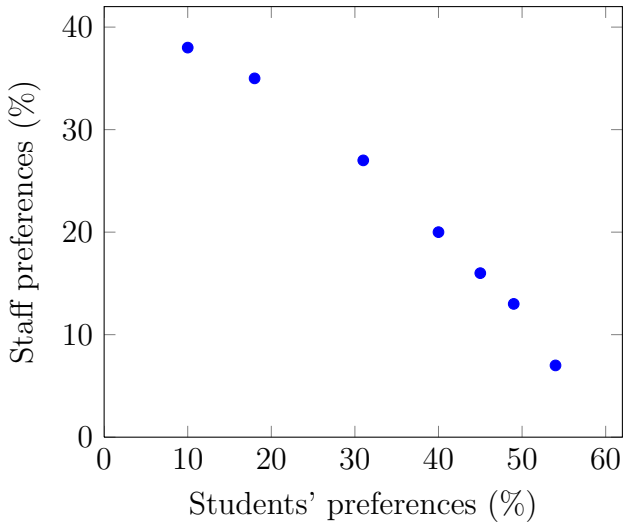

Figure 1.1: Example of trade-off set of solutions for a timetable problem

large. The subset represented on the right has low Uniformity, since there is a large gap between the two groups of solutions, while three of the solutions are very close to each other. Therefore, if the points represented in the Figure on the right are given to the decision maker, he is forced to choose extreme solutions that favour either the staff or the students. Another possibility is to use an indicator, such as the hypervolume [15] or the ϵ-indicator [9], which assign a scalar value to a set. (see Section 2.2 for an in-depth discussion of these indicators)

The aim of this thesis is to develop and analyse algorithms that find representative subsets of discrete sets of solutions, which is understood as a combinatorial optimisation problem. Two types of problems will be considered. The first problem is to select a subset of the trade-off set satisfying some properties, which may be used, for example, to present a small number of representative solutions to a decision maker. The second problem is to select a subset of solutions with respect to a reference set. This may be useful, for instance, in evolutionary algorithms, when selecting the individuals for the next generation. In either case, we assume that the desired cardinality of the subset is given *a priori* and that we want to obtain a subset with some guaranteed properties, such as Coverage, Uniformity and/or ϵ-indicator.

In the following paragraph, we introduce the notation and definitions that are needed to understand the thesis. A multiobjective combinatorial optimisation problem consists of a pair (X, f), where X is a discrete set composed of all feasible solutions, and $f = X \mapsto \mathbb{R}^d$ is the (vector) objective function, with d real-valued objectives. The image of X in the objective space is denoted by Y. The goal of these problems is to find the solution $x \in X$ that maximises $f(x)$ (w.l.o.g we assume maximisation of the d objectives). An element $x \in X$ is denoted solution, and $f(x)$ is its objective vector, or simply vector. For two solutions $x, x' \in X$, we say x weakly dominates x' or $f(x) \geq f(x')$ if $f(x)$ is greater or equal than $f(x')$ for every objective. Additionally, if $f(x) \geq f(x')$ and $f(x) \neq f(x')$, we say x dominates x' or $f(x) > f(x')$. If, for $x \in X$, there is no $x' \in X$ such that $f(x') > f(x)$, we say that x is efficient and $f(x)$ is non-dominated. The set $Y^{ND} \subset \mathbb{R}^d$ denotes the set of all non-dominated vectors, named non-dominated set, and X^E denotes the set of all efficient solutions, named efficient set. We also say that set $Y^i = Y \setminus \bigcup_{j=1}^{i-1} Y^j$ is the i-th non-dominated front, where $Y^1 = \{y \mid \nexists\, y' \geq y,\ y, y' \in Y\}$.

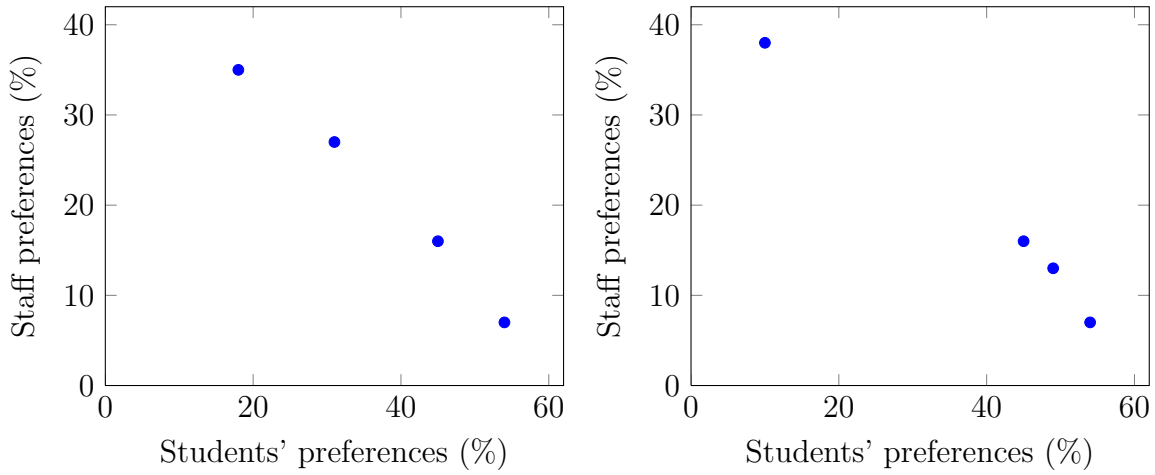

Figure 1.2: Two subsets of points with good Uniformity (left) and bad Uniformity (right)

The two problems mentioned above can be defined more formally as follows:

Let $Y^R \subseteq Y$ be a set of mutually non-dominated vectors, that is, $y \not\geq y'$ and $y' \not\geq y$ holds for all $y, y' \in Y^R$.

- **Problem 1:** Find a subset $Y^* \subseteq Y^R$, $|Y^*| = k$, where k is given, which optimises a given property.
- **Problem 2:** Let $Y' \subseteq Y$ be a set of mutually non-dominated vectors, satisfying $\forall\, y' \in Y', \exists\, y^R \in Y^R$ s.t. $y^R \geq y'$. We want to find a subset $\bar{Y} \subseteq Y'$, $|\bar{Y}| = k$, where k is given, that is as *close* as possible to set Y^R, according to a given measure.

The main contribution of this thesis is the discussion of these problems and the proposal of algorithms to solve them. Regarding the second problem, we propose three improvements to a known algorithm that optimises the ϵ-indicator, prove that the original algorithm is correct, and discuss the general case of the problem for more than two dimensions.

For the first problem, and with regard to the ϵ-indicator, we adapt the same algorithm for that specific case, present an alternative version, which is useful for multiobjective formulations of the problem, and adapt a known dynamic programming algorithm to use the ϵ-indicator. Then, we present other formulations of the problem, using Uniformity, Coverage, and combinations of these with the ϵ-indicator. For each discussed problem, we propose new versions of the algorithm that solve these problems, and use adaptations of known dynamic programming approaches to solve the same problems.

The document is structured as follows. We start by reviewing the literature concerning subset selection methods in Chapter 2. Then, we present the methodology that is used in the remainder of the thesis, in Chapter 3. In Chapter 4 we present an algorithm for the general 2D subset selection problem using the ϵ-indicator proposed by Ponte et al. [9], together with our contributions: i) the proof of correctness of the algorithm, and ii) three improvements that allow us to reduce its time complexity.

In the following chapters, we discuss the first problem described above, also identified as representation problem, using several properties to optimise. In Chapter 5, we introduce the representation problem, associated with the ϵ-indicator, and present two solutions, one of which is based on the work presented in Chapter 4, while the other uses the dynamic programming technique. In Chapters 6 and 7, we introduce two new

indicators two optimise, Coverage and Uniformity, respectively. Then, we discuss the applicability of the previous algorithms for these new indicators.

We then present some multiobjective properties, that is, problems for which we want to optimise combinations of indicators. In Chapter 8, we discuss the case where we consider the ϵ-indicator and Coverage simultaneously, and describe the necessary modifications to the algorithms. In Chapter 9, we use Uniformity instead of Coverage. Uniformity is not structurally similar to ϵ-indicator and Coverage, that is, it is calculated in a different way, and therefore this adaptation introduces new problems. Then, we combine these approaches and get a three objective problem, using ϵ-indicator, Coverage and Uniformity.

Finally, we present a general discussion and conclusions in Chapter 11. Additionally, we present some ideas for further work.

Chapter 2

State of the Art

Most of the approaches to multiobjective problems are based on evolutionary algorithms, since the idea of iteratively improving a set of solutions fits with the notion of trade-off set. These algorithms follow a common structure, starting with a set of solutions with fixed cardinality, called population. The algorithms work iteratively, applying biologically inspired operators, like mutation and crossover, to the elements of the population, called individuals. During the course of each iteration, single objective algorithms assign a fitness value to each individual to be used for selecting the elements that will form the population of the next iteration. However, the existence of multiple objectives implies that it is not possible to totally order the population. Multiobjective evolutionary algorithms use different techniques to overcome this problem.

In this chapter, we review subset selection techniques that have been used within population-based algorithms for multiobjective optimisation problems. These techniques select elements of the population for the next iteration and, therefore, are related to the main goals of this work. In Section 2.1 we present some algorithms that use the dominance relation to order the elements. In Section 2.2 we discuss two algorithms that use indicators to assign a quality value to each individual. Finally, in Section 2.3 we present some algorithms that use subset selection approaches in a more explicit manner.

2.1 Ranking-based Approaches

One of the earliest solutions to the problem of selecting the next population, used throughout some of the first multiobjective evolutionary algorithms, like MOGA [5], SPEA2 [14], and NSGA-II [4], is to order, in some manner, the solutions using a ranking function obtained from the dominance relation. For example, MOGA sorts the individuals according to the number of individuals in the population that dominate them, assigning a fitness value according to this ordering; SPEA2 takes into account the number of elements of the population that dominate and the elements that are dominated by an individual; NSGA-II divides the population in non-dominated fronts, assigning a rank according to the front an element belongs to.

In addition, the approaches above use techniques to maintain diversity: SPEA2 uses the distance to the k-th nearest neighbour, NSGA-II uses a crowding measure, and MOGA uses a fitness sharing technique. However, these approaches are used only to distinguish between individuals with the same ranking, either by giving a small weight to the di-

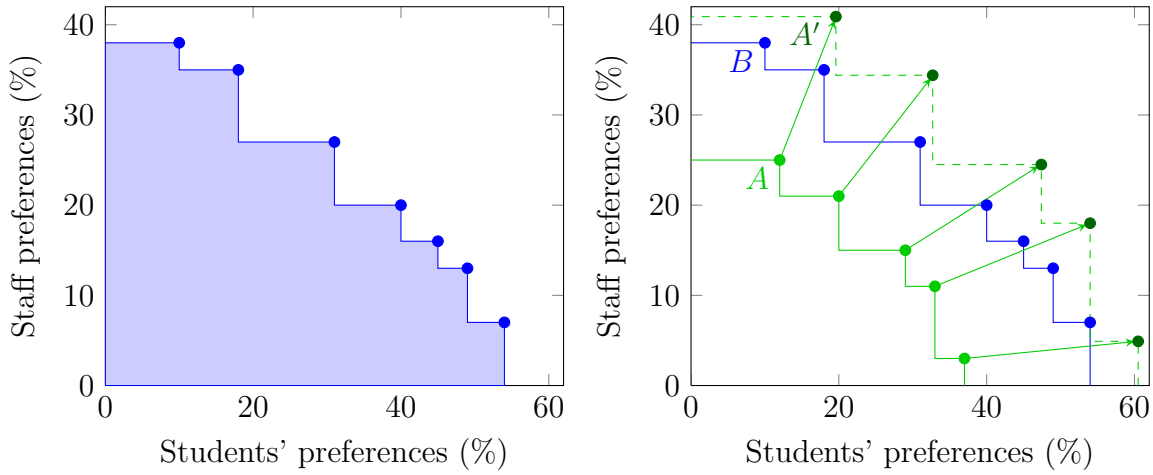

Figure 2.1: Illustration of the hypervolume and ϵ-indicator for the example of Figure 1.1

versity measure (SPEA2) or by comparing it only when the individuals have the same ranking (NSGA-II). Therefore, we conclude that, in these approaches, diversity is seen as a secondary goal.

Finally, MOGA presents an approach to incorporate decision making in the algorithm, by defining goals that specify "interesting" regions of the objective space. This approach allows a decision maker to use his knowledge of the problem to guide the search process, allowing the algorithm to obtain solutions that are more relevant to him.

2.2 Indicator-based Approaches

The indicator-based approaches measure the quality of a set by assigning it a real value. Then, this measure is used to guide the selection process. One of the most common indicators is the hypervolume. Given a set of vectors and a reference vector dominated by all the elements of the set, the hypervolume indicator is the measure of the region of the objective space defined by the set of vectors that dominate the reference vector and are dominated by at least a vector in the given set. In the left plot of Figure 2.1, the hypervolume indicator gives us the area of the shaded region, considering $(0, 0)$ as the reference point. Another example is the ϵ-indicator. We say a vector a ϵ-dominates a vector b or b is ϵ-dominated by a if $\epsilon \cdot a \geq b$. Given a set A and a reference set B, the ϵ-indicator is the minimum value ϵ such that every $b \in B$ is ϵ-dominated by an $a \in A$, that is,

$$I_\epsilon(A, B) = \max_{b \in B} \min_{a \in A} \epsilon(a, b),$$

where $\epsilon(a, b) = \max_{i \in \{1,\ldots,d\}} (b_i / a_i)$. In the right plot of Figure 2.1, an example of two sets, A (light green) and B (blue) are presented. Additionally, the figure illustrates that every element of B is dominated by an element of the set A' (dark green), obtained by multiplying the elements of set A by $\epsilon(A, B)$.

One of the advantages of these approaches is that they assign a quality value on a continuous scale, allowing us to use a scalar value to measure the difference between two sets [16]. However, indicators are often independent from the dominance relation, i. e. a greater indicator value (assuming maximisation) does not imply one of the sets is dominated. For instance, even if a set A has greater hypervolume than a set B, we cannot

conclude that A dominates B. Additionally, some indicators encompass information about the quality and diversity of the solutions, which means those approaches do not need to use a separate diversity technique [13].

Indicator-based approaches are used in various known algorithms, such as IBEA [13] and SMS-EMOA [2]. Both algorithms implement the hypervolume indicator and IBEA also presents a variation of the ϵ-indicator. Both these indicators are used to calculate a fitness value, using not only the element whose fitness is being calculated, but also information about the other individuals. In SMS-EMOA, the fitness of an individual is its individual contribution to the hypervolume, i.e. the hypervolume of the region dominated by the individual, but not by any other element in the population. IBEA presents a different notion of indicators, named binary indicators, that operate over two sets. For instance, the presented binary hypervolume indicator gives the volume of the space dominated by the second operand, but not by the first. However, the algorithm uses these operators with singleton sets, calculating fitness as the sum of the indicator values of an individual against the other individuals of the population. In other words, the algorithm calculates the indicator values for each pair of elements, and calculates the fitness as the sum of all the values where an element was used as a second operand.

While this is clearly an improvement in terms of flexibility and results [13], the individuals are still being compared using their one-dimensional fitness value. However, we have no guarantee that the greedy approach of choosing the best individual elements results in the optimal subset, according to a given indicator.

2.3 Subset Selection

The approach of subset selection consists in selecting the best subset with a given cardinality, according to some property of interest. Not many approaches use this technique: Bader [1] describes it for the hypervolume indicator, and Ponte et al. [9] uses the ϵ-indicator subset selection within a beam search algorithm.

Bader proposes an algorithm for biobjective problems that calculates the subset with largest hypervolume, given a fixed cardinality. The author also postulates that the subset selection problem, using the hypervolume, is NP-hard for more than two objectives, and presents a greedy alternative, similar to the approach described in SMS-EMOA. However, the author does not use this subset selection technique in any of his algorithms.

Regarding the ϵ-indicator, Ponte et al. propose a beam search algorithm for multiobjective optimisation, focusing on the biobjective $\{0, 1\}$ knapsack problem. The authors proposed a subset selection method for biobjective problems, within the beam search algorithm, using the ϵ-indicator. This approach is discussed in more detail in Chapter 4.

Chapter 3

Methodology

In the following chapters, we describe various algorithms that solve different problems. In order to compare the algorithms, we quantify their time complexity, using the random-access machine model, and then use experimental results to measure the performance of the different approaches for the same problems. Since all these algorithms are deterministic and return optimal solutions, we chose to do five repetitions for each instance of the problem, to account for slight fluctuations in running time. In these tests, running time is the only aspect of the execution which we compare, since the quality of the solutions for all the algorithms will be the same.

The tests were run in eight similar nodes of a cluster running the Sun Grid Engine for job management. The nodes where the tests were run are equipped with 16 GB of RAM and "Intel(R) Core(TM) i7-3770K" CPUs running at 1600MHz, with four jobs per node.

The source code for the implementations was written in C++ and compiled with GCC 4.4.3. C++ was chosen over C for its convenience, since its compiler is less strict. However, the Standard Template Library (STL) and the object-oriented features were not used. The only exception is the use of "vector" to store the solutions in the implementation of the algorithm described in Section 10.1.

Moreover, even though the tests were run in order to compare the performance of the different algorithms, they also served a second purpose of testing the correctness of the code. Since we know that the quality of the solutions must be the same for all the implementations for a given problem, then there must be an error if this does not happen. For any of the problems we describe, there are at least two different algorithms, that are written independently and using different ideas. Therefore, it is unlikely that the same error occurs in both implementations, in such a way that it is not detected. Furthermore, other more traditional tests were also used, such as modifying the input in predictable ways (changing one element, switching two elements) and comparing the output of small instances with the result obtained manually.

The sets of 2D vectors were generated randomly by sampling vectors from the intersection of a quarter circumference with the first quadrant. Then, each vector was rounded to obtain positive integer coordinates and the whole set was checked for dominated vectors. By generating more vectors than were required, the script accounts for the dominated vectors that are removed. Therefore, even if some vectors are removed, the desired cardinality is still obtained. If there are too many dominated vectors and the mechanism fails, exceptions are raised to ensure that the problem is noticed, and the

instance can be manually regenerated.

The tests done for each chapter are dependent of the parameters that influence the time complexity of the algorithms. In Chapter 4, there are two sets, A and B, whose size influences the complexity. Therefore, the tests focus on five different relations between the two variables $|A|$ and $|B|$:

- $|A| = |B|$, with $50 \leq |A| \leq 8000$
- $|A| = \log |B|$, with $50 \leq |B| \leq 8000$
- $|B| = \log |A|$, with $50 \leq |A| \leq 8000$
- $|A| = \sqrt{|B|}$, with $50 \leq |B| \leq 8000$
- $|B| = \sqrt{|A|}$, with $50 \leq |A| \leq 8000$

For this particular problem, we need to generate two sets A and B, such that the elements in B are not dominated by elements in A. Therefore, we use two different circumferences, such that they do not intersect in the first quadrant. Consequently, we assure that each of the elements of one set is dominated by at least an element of the other and thus satisfy the required condition.

For the remaining chapters, only one set is used as input, and parameter n denotes its size. Furthermore, some of the algorithms are influenced by the parameter k, representing the size of the desired subset. For the experimental tests in these chapters, two tests were run, allowing us to compare the influence of n and k individually.

The first test measures the influence of n in the running time. Therefore, several instances are generated for different values of n, while k is kept constant. The second test is similar, although the studied parameter is k. The various instances of the second test have different values of k, and n is kept constant.

In Chapters 5 and 7, parameter n varies between 50 and 2000 in the first test, with k taking the value 20. For the second test, the maximum value for n, 2000, is used, and k takes values between 50 and 1000. Chapters 8 and 9 use similar values. However, the maximum values of n and k are 1000 and 500, respectively. In Chapter 10, these values are reduced further to 100 and 50, respectively.

Chapter 4

Subset Selection using ϵ-indicator

Given two sets of vectors A and B, and k, the subset selection problem consists in finding a subset $R \subseteq A$, with $|R| = k$, which minimises the value of ϵ, such that each element of B is dominated by an element of the set R', obtained by multiplying each element of R by ϵ. Using the notation introduced in Chapter 2, our goal is to find:

$$\underset{\substack{R \subseteq A \\ |R|=k}}{\arg\min}\ I_\epsilon(R, B)$$

This problem can also be formulated using graph notation, since the problem of finding the minimum value of ϵ is related to the set covering problem in a bipartite network, where the two parts of the network are the sets A and B. Therefore, we may define $G = (A, B, E)$, where each edge $(a, b) \in E$, with $a \in A$ and $b \in B$ has weight $\epsilon(a, b)$.

In this chapter, Section 4.1 describes the algorithm proposed by Ponte et al. [9] to solve the bidimensional case of the problem described above. In Section 4.2, we show that the algorithm is correct, demonstrating some less trivial steps. In Section 4.3, we propose some improvements to the algorithm by Ponte et al., which allow us to reduce the time complexity of the original algorithm. In Section 4.5, we show that this approach cannot be easily extended for three or more dimensions.

4.1 Threshold Algorithm

Ponte et al. [9] proposed an algorithm that solves the subset selection problem for the bidimensional case, using the ϵ-indicator. This algorithm has two main phases: first, it finds the optimal value for the ϵ-indicator, ϵ_{opt}, and then uses it to find a subset R such that $I_\epsilon(R, B) = \epsilon_{opt}$.

Let $\epsilon_1, \epsilon_2, ..., \epsilon_{|A|\cdot|B|}$ be the sorted list of the weights of the edges in E (in non-decreasing order). For each value of ϵ_i, we consider the graph $G_i = (A, B, E_i)$, where E_i is the set of all the edges in E whose weight is less than or equal to ϵ_i. Consequently, each edge $(a, b) \in E_i$ means that $\epsilon_i \cdot a \geq b$ or, in other words, $\epsilon(a, b) \leq \epsilon_i$. An example is given in Figure 4.1, where we consider a value of $\epsilon_i = 2$. In this case, each element $a \in A$ is connected to an element $b \in B$ if $2a \geq b$.

We know that ϵ_{opt} takes the value of one of the edge weights, since these are the

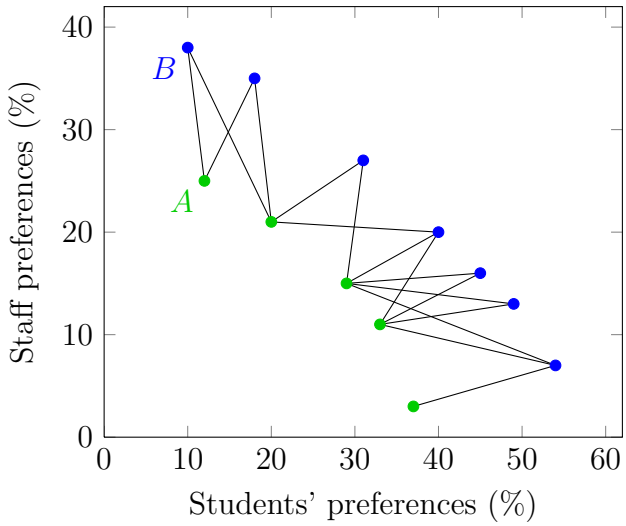

Figure 4.1: Example of a graph G_i for the example given in Figure 2.1

minimum values for each configuration of edges. Consequently, we simply have to test each value of ϵ_i and find the minimum value that allows us to get an appropriate subset $R \subseteq A$.

In order to accomplish its goal, the algorithm tests each ϵ_i by finding the smallest subset $R_i \subseteq A$ such that each element $b \in B$ has an edge connecting it to an element $r \in R_i$ in the graph G_i. If $|R_i| > k$, then the edges of G_i are not sufficient to obtain the desired subset. Consequently, we need to use more edges, which means that $\epsilon_{opt} > \epsilon_i$. On the other hand, if $|R_i| < k$, we may add any element $a \in A \setminus R_i$ to R_i, since the ϵ-indicator $I_\epsilon(R_i, B)$ is calculated by finding, for each $b \in B$, the minimum value of $\epsilon(r, b)$, for every $r \in R_i$. Therefore, this minimum value does not increase by adding elements to R_i, which means we may add elements to R_i without increasing $I_\epsilon(R_i, B)$. After finding the value of ϵ_{opt}, it is sufficient to find the smallest subset $R_i \subseteq A$, whose cardinality is at most k. Figure 4.1 illustrates a graph where every element of B is connected to either the first or third element of A, counting from left to right and, therefore, we have $|R_i| = 2$. If, for instance, we wanted only one element, the edges in the graph presented would be insufficient, and we would need to consider a larger value of ϵ_i, which would imply more edges. If, on the other hand, we had $k = 3$, then we would test lower values for ϵ_i, in order to determine if a better solution exists.

We can also reduce the number of values of ϵ_i to test by considering upper and lower limits. The lower limit is given by $I_\epsilon(A, B)$. Even though $|A| > k$, it is not possible to obtain a lower ϵ value, since adding elements to a set will not increase the ϵ-indicator and we can obtain A by adding elements to any subset. For the upper limit, we can consider the ϵ values for each singleton set, since these subsets are valid solutions to our problem. Then, we select the minimum ϵ found as the upper limit. Formally, we have the lower limit $\epsilon_m = I_\epsilon(A, B)$ and the upper limit $\epsilon_M = \min_{a \in A} I_\epsilon(\{a\}, B)$.

Based on the idea described above, the algorithm only needs to find the set R_i, given a value of ϵ_i. This subproblem is similar to a particular case of the set covering problem, which can be solved using the algorithm proposed by Schöbel [11]. For this particular case, the adjacency matrix of the graph satisfies the "consecutive ones property", i.e. the ones in every row of the adjacency matrix are consecutive. In this case, we consider that each row corresponds to an element $b \in B$ and each column to an element $a \in A$. If we

sort the rows and columns of the matrix according to the first coordinate, this statement is true, as is demonstrated in Section 4.2.

Since the algorithm proposed by Schöbel has linear time complexity and receives, for each row, the first and last column with the value one, we just need to find these indexes. The paper by Ponte et al. uses the naive approach to do this by going over each row to find where the matrix has a value of one, with time complexity $\mathcal{O}(|A| \cdot |B|)$.

According to the algorithm, the procedure to find R_i must be performed for several values of ϵ_i. In order to reduce the number of values of ϵ_i to test, the authors suggest the use of binary search to find the value of ϵ_{opt}. This is possible since if $|R_i| > k$, we know that $\epsilon_{opt} > \epsilon_i$ and if $|R_i| \leq k$, then $\epsilon_{opt} \leq \epsilon_i$, which means we can use the value of $|R_i|$ to guide the binary search. However, we still need to ensure the values of $|R_i|$ are ordered, which we can only do by sorting the values ϵ_i, which has time complexity $\mathcal{O}(|A| \cdot |B| \log(|A| \cdot |B|))$.

Considering this structure, we may divide the algorithm in two parts: the preprocessing, which consists in sorting the values of ϵ_i, and whose time complexity we denote by $T_P(A, B)$, and the search itself, consisting in executing the procedure to solve the set covering problem, with time complexity $T_C(A, B)$, repeated over $T_S(A, B)$ values of ϵ_i, resulting in a time complexity of $\mathcal{O}(T_P(A, B) + T_S(A, B) \cdot T_C(A, B))$. For the algorithm proposed by Ponte et al., we have the complexities given by: $T_P(A, B) = \mathcal{O}(|A| \cdot |B| \log(|A| \cdot |B|))$; $T_S(A, B) = \mathcal{O}(\log(|A| \cdot |B|))$; $T_C(A, B) = \mathcal{O}(|A| \cdot |B|)$. Therefore, the global time complexity for the algorithm is $\mathcal{O}(|A| \cdot |B| \log(|A| \cdot |B|))$. However, there are some improvements that can be done, leading to different time complexities. These possibilities will be described in Section 4.3.

4.2 Correctness

Ponte et al. do not show that the algorithm described above is correct. The algorithm is based on the adjacency matrix having the "consecutive ones property". Therefore, in this section we will prove this statement, guaranteeing its correctness. We start by demonstrating that every row of the adjacency matrix has at least a one, and move on to proving all the ones in the rows of the adjacency matrix are consecutive.

Proposition 4.2.1. *If $\epsilon_i \geq \epsilon_m$, there can be no row of the adjacency matrix of G_i consisting completely of zeroes.*

Proof. By definition, $\epsilon_m = I_\epsilon(A, B)$, i.e. it is the value of the ϵ-indicator if we select the set A as our solution to the problem. If we consider a value $\epsilon_i \geq \epsilon_m$, only edges with weight greater than ϵ_i, and consequently greater than ϵ_m, will be removed. Therefore, the edges of G_m, the graph corresponding to ϵ_m, will be present in G_i, which means every element $b \in B$ has at least an incident edge, or equivalently, a value of one in the corresponding row of the matrix. □

Proposition 4.2.2. *The adjacency matrix of the graph G_i corresponding to the value $\epsilon_i \geq \epsilon_m$ has the "consecutive ones property" (C1P).*

Proof. To prove this property holds, we proceed by contradiction, by saying that for some row, corresponding to an element $b \in B$, there are two ones that are not consecutive, i.e. there is a zero between them. Notice that this is the only possibility, since a row cannot

consist only of zeroes, by Proposition 4.2.1. Formally, there are $x, y, z \in A, x_1 < y_1 < z_1$ such that $\epsilon(x, b) \leq \epsilon_i$, $\epsilon(y, b) > \epsilon_i$ and $\epsilon(z, b) \leq \epsilon_i$. Since $\epsilon_i \geq \epsilon(x, b) = \max(b_1/x_1, b_2/x_2)$, then $b_1/x_1 \leq \epsilon_i$. Since $x_1 < y_1 \Leftrightarrow b_1/x_1 > b_1/y_1$ and $b_1/x_1 \leq \epsilon_i$, then $b_1/y_1 \leq \epsilon_i$. Following a similar reasoning for y, z, and knowing that $y_2 > z_2$ (or y would be dominated by z), we conclude $b_2/y_2 \leq \epsilon_i$. Finally, since we have that $\epsilon(y, b) = \max(b_1/y_1, b_2/y_2)$ and both $b_1/y_1 \leq \epsilon_i$ and $b_2/y_2 \leq \epsilon_i$, then $\epsilon(y, b) \leq \epsilon_i$, which leads to a contradiction, finishing our proof that the matrix has the "consecutive ones property". □

4.3 Improvements on the Algorithm

4.3.1 Set Covering

The procedure that finds the smallest subset R_i for a given ϵ_i is executed over some values of ϵ_i, since it is used to guide the search for the value of ϵ_{opt}. However, this procedure has a time complexity of $T_C(A, B) = \mathcal{O}(|A| \cdot |B|)$, which means that even by removing the preprocessing step, we would still have a global time complexity of $\mathcal{O}(|A| \cdot |B| \log(|A| \cdot |B|))$. In the following, a new search for the beginning and end of the consecutive ones is proposed, with an improved time complexity of $T_C(A, B) = \mathcal{O}(|B| \log |A|)$.

In order to find the entries of a row that have ones, we work with the weighted matrix of the original network G. Since the graph G_i is obtained from G by removing the edges whose weight is greater than ϵ_i, we simply have to find the first and last values that are at most ϵ_i. Also, since $\epsilon(a, b) = \max(b_1/a_1, b_2/a_2)$, each row is bitonic, as a_1 increases and consequently, a_2 decreases, with the value of $\epsilon(a, b)$ decreasing while $b_1/a_1 > b_2/a_2$ and increasing when $b_1/a_1 < b_2/a_2$. Therefore, if the location of the minimum for each row is known (which the algorithm may store when building the sequence ϵ_i), the bitonic row can be split into two sorted subsequences, which allows binary search to be applied. This means we can find the indexes in each of the two subsequences, where $\epsilon(a, b)$ is as large as possible, while being at most ϵ_i, with time complexity $\mathcal{O}(\log |A|)$. Since we need to repeat this operation for each of the $|B|$ rows, we have a time complexity of $\mathcal{O}(|B| \log |A|)$ for the procedure of finding the subset R_i.

Note that, although we have $T_C(A, B) = \mathcal{O}(|B| \log |A|)$, our time complexity is still given by $\mathcal{O}(|A| \cdot |B| \log(|A| \cdot |B|))$, since the preprocessing step of sorting the values of ϵ_i dominates the time complexity of the second part of the algorithm.

4.3.2 Merge Sort

After applying the improvement described in Section 4.3.1, the time complexity of the search is reduced, and therefore the time complexity of sorting the values of ϵ_i dominates the time complexity of the search. Therefore, in order to improve the algorithm, we must improve the sort (or remove it), which corresponds to an improvement of $T_P(A, B)$.

Due to the bitonic nature of the rows, we are able to sort each row in linear time, by performing a merge of the increasing and decreasing subsequences. This allows the algorithm to sort the list of the values of ϵ_i by first sorting each row in $\mathcal{O}(|B| \cdot |A|)$ ($|B|$ rows with $|A|$ elements) and then merging all the rows in $\mathcal{O}(\log |B|)$ iterations, building sorted lists of size $2|A|$ in the first iteration, then $4|A|$, and so on until the entire

list is sorted. Since with each iteration, the size of the lists that are merged doubles, we need $\mathcal{O}(\log |B|)$ iterations in order to have the entire list with size $|B| \cdot |A|$ sorted. Consequently, this sorting algorithm has time complexity of $\mathcal{O}(|A| \cdot |B| \log |B|)$, since it has to run through the entire list in each iteration.

Similarly to the original version, the algorithm then uses binary search to find the optimum value of ϵ_i. Since binary search divides the list in two at each iteration and the list has $|A| \cdot |B|$ elements, the algorithm only needs $\log(|A| \cdot |B|)$ iterations in order to reduce the list to a single element. However, unlike the "common" binary search algorithm, the cardinality of $|R_i|$ is used to guide the search, so the procedure to find the set R_i, for a given value of ϵ_i, is executed. Consequently, in each of the $\log(|A| \cdot |B|)$ iterations, a procedure with time complexity $\mathcal{O}(|B| \log |A|)$ is called, which means searching for the optimum value of ϵ has $\mathcal{O}(|B| \log |A| \log(|A| \cdot |B|))$ time complexity.

Concluding, the preprocessing step is now $T_P(A, B) = \mathcal{O}(|A| \cdot |B| \log |B|)$, which improves the global time complexity to $\mathcal{O}(|A| \cdot |B| \log |B| + |B| \log |A| \log(|A| \cdot |B|))$, or, due to the properties of logarithms, $\mathcal{O}\left(|A| \cdot |B| \log |B| + |B| \log^2 |A| + |B| \log |A| \log |B|\right)$. Also, since $|A|$ asymptotically dominates $\log |A|$ and $\log^2 |A|$, then $|A| \cdot |B| \log |B|$ dominates $|B| \log |A| \log |B|$ and $|B| \log^2 |A|$. Consequently, we have a simplified time complexity of $\mathcal{O}(|A| \cdot |B| \log |B|)$.

Although this time complexity is better than the naive version, reducing the time complexity of the sort or simply removing it could allow us to reduce the time complexity of the whole algorithm. However, not sorting the list means we cannot use a binary search approach, so a different approach must be used.

4.3.3 Fractional Cascading

An alternative to sorting and using binary search over the entire sorted list is to do binary search over each individual row. Even though it is still necessary to sort the rows, it takes time complexity $\mathcal{O}(|A| \cdot |B|)$, since the rows have a bitonic nature and we only need to merge the decreasing and increasing subsequences.

A technique proposed by Chazelle and Guibas [3], named Fractional Cascading, allows us to speedup the algorithm by doing a single binary search, with a preprocessing step of $\mathcal{O}(|A| \cdot |B|)$. Preprocessing the data enables the algorithm to do the search in all rows after the first using just one test, which means we only need to do a binary search in the first row. Also, being based on binary search, we are able to adapt the idea we used in the binary search algorithm, which allows the algorithm to search for the optimum value of ϵ, instead of a fixed value.

The preprocessing step consists in creating a new matrix, based on the weighted matrix of the graph G, in which each row is assigned an element $b \in B$ and each column an element $a \in A$. This matrix is composed of $|B|$ rows, but unlike the original matrix, each row may have up to $2|A|$ elements, including the $|A|$ elements of the original row, merged with some elements from rows below. Each element of the rows stores a value (in our case, an ϵ_i), and two integer numbers. The first of these numbers stores the position of that value within the original row, allowing the algorithm to find the position in the original matrix, while the second number tells us where to find that value in the next row, which will enable the algorithm to continue the search in the next row, without searching from the beginning. When using this technique to find the optimum ϵ value, we do not need to know where that value is in the original matrix, so we may discard the

1.9	3.5	5.2
1.2	2.5	4.6
1.3	4.4	6.2

1.9(1)	2.5(1)	3.5(2)	4.6(3)	5.2(4)
1.2(0)	2.5(1)	4.4(1)	4.6(2)	
1.3(0)	4.4(0)	6.2(0)		

Table 4.1: Example of weighted matrix (with sorted rows) and the processed matrix

first number, and keep only the values and positions in the next row.

To build the matrix, the algorithm starts by sorting each row of the original matrix, similarly to what is done in the merge sort improvement. Then, the algorithm starts with the last row and copies it to the new matrix. Being the last row, we do not use the position numbers in the next row, so they do not need to be initialised. The algorithm then moves on to the previous row, and merges the values of the original row with the values of the next row of the new matrix that have an even position. In other words, for the next-to-last row, it takes the values with even position (second, fourth, ...) of the last row and merges them with the values in the next-to-last row of the original matrix, while at the same time initialising the position numbers. These position numbers, for the values that are copied from the last row, are simply the position of that value in the last row, while, for the other values in the next-to-last row, the number saved is the index of the smallest value in the last row that is greater or equal to the value in the next-to-last row. This procedure is repeated for every row, starting from the next-to-last and finishing with the first.

In Table 4.1, an example of the preprocessing step is presented. Given the matrix with sorted rows, the algorithm copies the third row to the new matrix. For the second row of the new matrix, the original row is merged with the second element of the last row, and the position numbers indicate the position where that element would be in the next row. For instance, the position number of the second entry in the second row is 1 since 2.5 is between the values with indexes 0 and 1 in the last row. Repeating this process, we obtain the first row of the new matrix by merging the first row of the original matrix and the second and fourth elements of the second row of the new matrix. Note that since 5.2, in the first row, is larger than any value of the second row, the corresponding index is 4, since that value would occupy the last position in the second row.

Using this new matrix, the search is done by searching for the value in the first row, using binary search, which allows the algorithm to find the position of the optimum value in the first row, or the smallest value greater than the optimum. Since in our case, the algorithm is not searching for a specific value of ϵ_i, the algorithm finds the smallest value of ϵ_i that yields a feasible solution, i.e. that results in a subset with at most k elements. Given the position of the value in the first row, the algorithm reads the number s that represents the position of that value in the next row. Then, the Fractional Cascading technique guarantees that the best value of the next row is on positions s or $s-1$. Also, since the value in the position s is greater or equal than the value we found, we know that it also represents a valid solution, so we only need to test the value in the position $s-1$ to check if it gives us a better solution. Repeating this for every row, we only need to perform a binary search in the first row and check a value for each other row to find the optimum value of ϵ.

This approach uses more space than the other versions of the algorithm, since it needs to build a matrix of size $2|A|$ by $|B|$, storing an ϵ value and an integer in each entry. However, the spatial complexity is $\mathcal{O}(|A| \cdot |B|)$, so even if, in practice, this version uses more

space, the spatial complexity is the same as in the other versions. Concerning the time complexity, this algorithm is composed of a preprocessing step and the search with both contributing to the final complexity. For each row of the new matrix, the preprocessing step merges a row of the original matrix, with size $|A|$ and a row of the new matrix, with size at most $2|A|$. Consequently, the merge operation has $\mathcal{O}(|A|)$ time complexity. Since copying the last row from the original to the new matrix also has a time complexity of $\mathcal{O}(|A|)$, executing these operations $|B|$ times has $\mathcal{O}(|A| \cdot |B|)$ time complexity. For the search, the algorithm uses binary search in the first row, requiring $\mathcal{O}(\log |A|)$ tests, with $\mathcal{O}(|B|)$ more tests, one for each row. Therefore, this version of the algorithm changes the complexities of the preprocessing step and the search to $T_P(A, B) = \mathcal{O}(|A| \cdot |B|)$ and $T_S = \mathcal{O}(|B| + \log |A|)$. Since testing each value of ϵ_i has $T_C(A, B) = \mathcal{O}(|B| \log |A|)$ time complexity, the overall search time complexity $\mathcal{O}((|B| + \log |A|) \cdot |B| \log |A|)$.

Combining the preprocessing step and search itself, we get a time complexity of $\mathcal{O}(|A| \cdot |B| + (|B| + \log |A|) \cdot |B| \log |A|)$. Simplifying this expression, we get a time complexity of $\mathcal{O}\left(|A| \cdot |B| + |B|^2 \log |A| + |B| \log^2 |A|\right)$, and since $|A|$ dominates $\log^2 |A|$, $\mathcal{O}(|A| \cdot |B| + |B|^2 \log |A|)$.

If we have $|B| > |A|$, this does not seem like an improvement, since it will be worse than the previous versions. However, unlike the procedure to find R_i, the Fractional Cascading technique is not aware of the specific problem being solved. Therefore, by simply doing the transposition of the weighted matrix and doing the search over the rows of this new matrix (columns of the original matrix), we change the time complexity, since $T_S(A, B) = \mathcal{O}(|B| + \log |A|)$ term becomes $T_S(A, B) = \mathcal{O}(|A| + \log |B|)$. Since the columns have the same bitonic nature as the rows, we can still apply the same algorithm, but scanning over the columns of the original matrix, when doing the preprocessing step. Consequently, using the transposition we get the time complexity $\mathcal{O}(|A| \cdot |B| + (|A| + \log |B|) \cdot |B| \log |A|)$. This expression can be expanded, obtaining $\mathcal{O}(|A| \cdot |B| + |A| \cdot |B| \log |A| + |B| \log |A| \log |B|)$ and since $|A| \cdot |B|$ is dominated, we have a final time complexity of $\mathcal{O}(|A| \cdot |B| \log |A| + |B| \log |A| \log |B|)$.

4.4 Complexity and Results

In Table 4.2, the complexities of the different versions of the algorithm are presented. We consider three different cases: i) General corresponds to the most general scenario; ii) $|A| = |B|$ arises from the case where $A = B$, which is related to the first goal of this work; and iii) $|A|$ is constant, which is usually the case that arises in evolutionary algorithms with a population of fixed cardinality. The versions of the algorithm described are: Original, the algorithm proposed by Ponte et al. [9]; Imp. 1, which is presented in

Version	**General**	$\|\mathbf{A}\| = \|\mathbf{B}\|$	$\|\mathbf{A}\|$ **is constant**
Original	$\mathcal{O}(\|A\| \|B\| \log(\|A\| \|B\|))$	$\mathcal{O}\left(\|B\|^2 \log \|B\|\right)$	$\mathcal{O}(\|B\| \log \|B\|)$
Imp. 1	$\mathcal{O}(\|A\| \|B\| \log(\|A\| \|B\|))$	$\mathcal{O}\left(\|B\|^2 \log \|B\|\right)$	$\mathcal{O}(\|B\| \log \|B\|)$
Imp. 2	$\mathcal{O}(\|A\| \|B\| \log \|B\|)$	$\mathcal{O}\left(\|B\|^2 \log \|B\|\right)$	$\mathcal{O}(\|B\| \log \|B\|)$
Imp. 3a	$\mathcal{O}\left(\|A\| \|B\| + \|B\|^2 \log \|A\|\right)$	$\mathcal{O}\left(\|B\|^2 \log \|B\|\right)$	$\mathcal{O}\left(\|B\|^2\right)$
Imp. 3b	$\mathcal{O}(\|A\| \|B\| \log \|A\| + \|B\| \log \|A\| \log \|B\|)$	$\mathcal{O}\left(\|B\|^2 \log \|B\|\right)$	$\mathcal{O}(\|B\| \log \|B\|)$

Table 4.2: Summary of the time complexities of the different versions

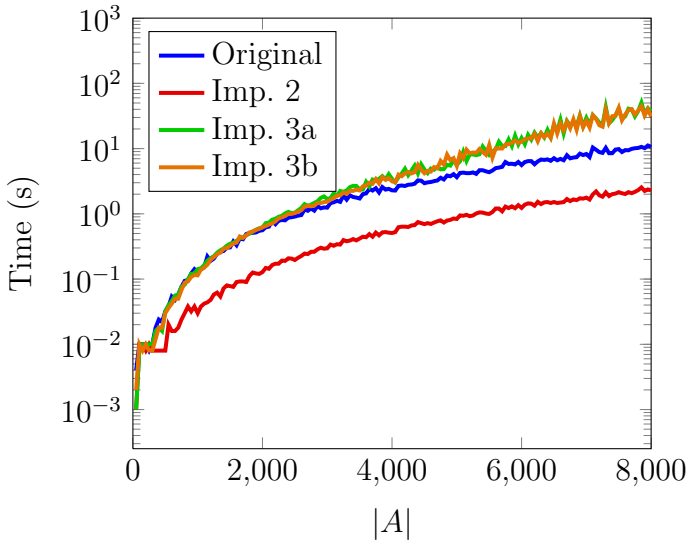

Figure 4.2: Running time for the four approaches with $50 \leq |A| = |B| \leq 8000$, $k = 20$

Section 4.3.1; Imp. 2, which is presented in Section 4.3.2; Imp. 3a and Imp. 3b, which are the first and second versions presented in Section 4.3.3, respectively.

Starting with the general case, we can see that the first improvement does not reduce the time complexity of the algorithm, even though it reduces the time complexity of the search, allowing us to improve the time complexity by improving the preprocessing step. The next three versions of the algorithm have different complexities, and are in fact incomparable, which means that we do not have an implementation that is better for all the cases. For example, if we consider $|B| > |A|$, then improvement 3b is better than any other version, since the term $|B| \log |A| \log |B|$ will be dominated by any of the other complexities, and $|A| \cdot |B| \log |A|$ will be smaller than the other terms.

Comparing the different versions for the particular cases when $|A| = |B|$ and $|A|$ is constant, gives us a different view on the complexities. For $|A| = |B|$, all of the versions have the same time complexity, $\mathcal{O}(|B|^2 \log |B|)$. On the other hand, we may consider $|A|$ as a constant, which is the case of the beam search proposed by Ponte et al. [9]. In this case, all the versions have a time complexity of $\mathcal{O}(|B| \log |B|)$, except for improvement 3a that has $\mathcal{O}(|B|^2)$ time complexity. This improvement is one of the alternatives proposed in Section 4.3.3, which is more suitable for the case $|A| > |B|$. Therefore, this approach would not be recommended, since we are considering $|A|$ a constant, which means that $|A|$ is asymptotically less than $|B|$.

4.4.1 Experimental Results

The complexities of all the versions of the algorithm depend on the size of the sets A and B. Therefore, these were the dimensions we wanted to study with the experiments. Since different algorithms may perform differently depending on the relation between the sizes of A and B, we decided to study five different cases, expressing different relations between $|A|$ and $|B|$.

We start by looking at the simplest case, where we have $|A| = |B|$, and then consider the cases in which $|A| = \log |B|$, $\log |A| = |B|$, $|A| = \sqrt{|B|}$ and $\sqrt{|A|} = |B|$. We refer to Chapter 3 for a more detailed discussion of the experimental methodology.

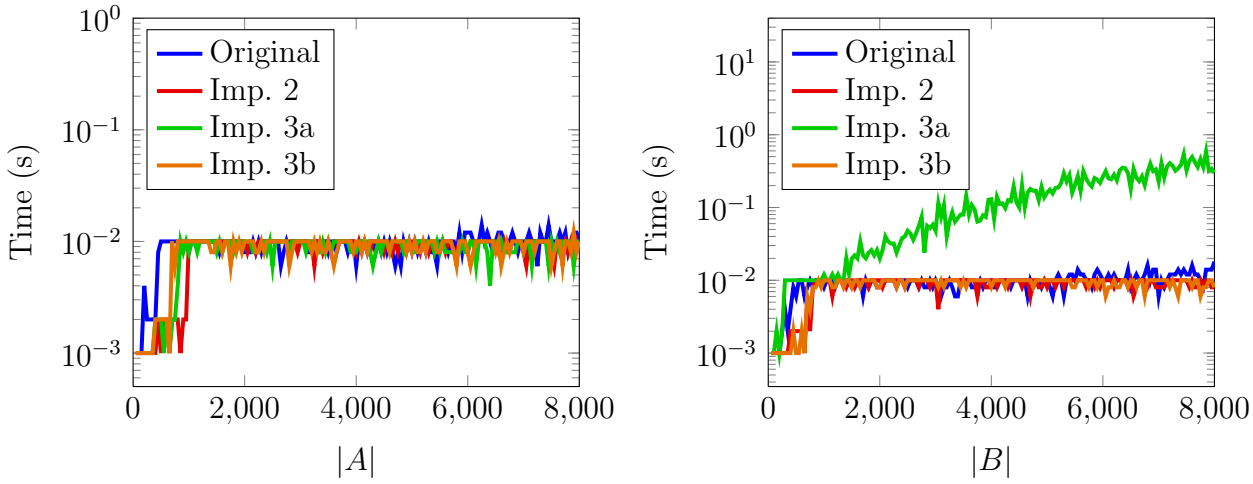

Figure 4.3: Running time for the cases $|B| = \log |A|$ (left) and $|A| = \log |B|$ (right)

The results for the case $|A| = |B|$ are given in Figure 4.2. As expected, the second improvement, using merge sort, performs better than the original version. However, either of the Fractional Cascading versions are much slower than the original version. This can be explained by the increased overhead of creating the matrix and the increased number of values of ϵ-indicator to test, which is $\mathcal{O}(|B| + \log |A|)$ or $\mathcal{O}(|A| + \log |B|)$ for the Fractional Cascading versions, compared to $\mathcal{O}(\log(|A| \cdot |B|))$ for the original and merge sort versions.

Looking at these results, the merge sort version is clearly the fastest when considering these conditions, being four times as fast as the original version. However, this version may not be the better in every case, considering that the two versions of Fractional Cascading are aimed at the cases where $|A| < |B|$ and $|A| > |B|$, which means these algorithms may benefit from having the set A much smaller than B.

We now look at the cases where $\log |A| = |B|$, $|A| = \log |B|$, whose results are presented in Figure 4.3. These results are not as clear as in the previous test, but they nevertheless tell us two important things. First of all, it is clear that the sizes of both A and B influence greatly the performance of the different versions of the algorithm, allowing the algorithms to run in some milliseconds when one of the sets has less than 50 elements. The second important remark is that Improvement 3a is an exception, as it is extremely slow, even if we consider a small number of elements of A with a large value of $|B|$. In fact, while Improvement 3b performs reasonably well in both these tests, Improvement 3a only performs well when B is relatively small. This is explained by its time complexity, which includes a $|B|^2$ term.

Finally, we have the two last tests, which confirm the results until now and are presented in Figure 4.4. These tests consider a quadratic relation between the sizes of A and B, with $|B| = \sqrt{|A|}$ and $|A| = \sqrt{|B|}$, respectively. In the first test, all three improvement versions are better than the original version, with Improvement 3a and the merge sort version having similar performance, better than Improvement 3b. In the second test, the opposite happens, with the merge sort version and Improvement 3b sharing similar performance, better than the original version and Improvement 3a. Once more, Improvement 3a is the worst version, with the $|B|^2$ factor causing the algorithm to slow down, even with a low $|A|$.

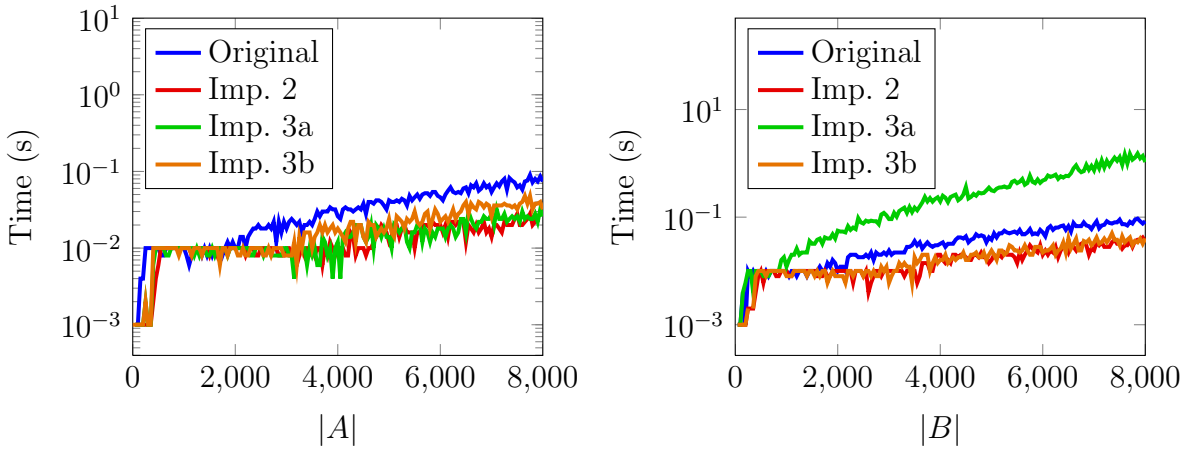

Figure 4.4: Running time for the cases $|B| = \sqrt{|A|}$ (left) and $|A| = \sqrt{|B|}$ (right)

In conclusion, we can see that the Fractional Cascading versions behave poorly when the sizes of A and B are similar, and perform well on the respective cases. However, Improvement 3a performs poorly when $|B|$ is large, whereas Improvement 3b still performs reasonably well when $|A|$ is large, if $|B|$ is small. Also important is the fact that the merge sort version is one of the best versions in every test, and is never clearly surpassed by any other version.

4.5 Extension to 3D

In the bidimensional case described above, the adjacency matrix of the graph has the "consecutive ones property", since by sorting the vectors using one coordinate, they also become sorted by the other coordinate (in the reverse order). However, this property may not hold for more than two dimensions, as we state in Proposition 4.5.1, which means that the same algorithm cannot be applied to the generic case.

Proposition 4.5.1. *For three or more dimensions, the adjacency matrix is not guaranteed to satisfy the "consecutive ones property", even by reordering its rows or columns.*

Proof. We will consider, without loss of generality, the three-dimensional case, since for additional dimensions, the vectors may have a fixed value, which will not alter the ϵ-indicator and consequently, the adjacency matrix. Let $B = \{(3,5,2),(4,4,8),(5,3,2)\}$ and $A = \{(1,3,2),(2,2,1),(3,1,2)\}$. If we denote by b_1, b_2, b_3 the elements of B and a_1, a_2, a_3 the elements of A, in the presented order, then the weighted matrix for this problem is given in Table 4.3. For $\epsilon_i = 4$, the adjacency matrix will be composed of ones, except for the values of the antidiagonal (in bold), which will be zero, since the values of the weighted matrix in that positions are larger than ϵ_i. Therefore, we have three zeroes, one in each column and one in each row, which means that independently of the ordering of the rows or columns, one of the rows will have a zero in the central position, with two non-consecutive ones at the leftmost and rightmost positions. □

Consequently, we proved that there are instances of the problem that cannot be solved using this approach, since the adjacency matrix does not have the required properties.

	$\mathbf{a_1}$	$\mathbf{a_2}$	$\mathbf{a_3}$
$\mathbf{b_1}$	3	2.5	**5**
$\mathbf{b_2}$	4	**8**	4
$\mathbf{b_3}$	**5**	2.5	3

Table 4.3: Weighted matrix for the proof of Proposition 4.5.1

Therefore, the algorithm described for the bidimensional case cannot be easily extended for three or more dimensions.

Chapter 5

Representation Problem using ϵ-indicator

Given a set of vectors B and k, the representation problem consists in finding a subset $R \subseteq B$, with $|R| = k$, that optimises a given property. In this case, we want to minimise the ϵ-indicator, that is, find a subset for which the value of the ϵ-indicator is smallest, such that each element of B is ϵ-dominated by an element of R. Using the notation introduced in Chapter 2, we have the goal given by:

$$\underset{\substack{R \subseteq B \\ |R|=k}}{\arg\min}\ I_\epsilon(R, B)$$

This is the first goal of this thesis, as described in Chapter 1. This problem can be solved using an adaptation of the Threshold Algorithm, discussed in Chapter 4. In Section 5.1 we present this algorithm, and an alternative version, which is used in later chapters, even though it is less efficient. In Section 5.2, we present a different algorithm, using dynamic programming.

5.1 Threshold Algorithm

The binary search algorithm presented in this section is an adaptation of the one previously described in Chapter 4. Given the large of range of versions for the subset selection algorithm, and given that the optimised version with merge sort was the fastest version overall, we decided to use it for the representation problem.

First of all, we use only one set, that is, we consider $A = B$, which means the weight matrix is now a square matrix with ones on the diagonal. Since the value of the ϵ-indicator is at a minimum on the diagonal, it is now trivial to calculate the minimum for each row.

Also, since we proved that the weight matrix has the consecutive ones property for the general case (see Section 4.2), the proof also applies to this specific case. Using $n = |A| = |B|$, we have a complexity of $\mathcal{O}(n^2 \log n)$.

5.1.1 Alternative Algorithm

Even though the algorithm described in the previous section is efficient, it is hard to adapt its structure to other indicators, in order to use it in the multiobjective version of the problem (see Chapter 9). Therefore, we present an alternative to the set covering procedure presented in Chapter 4, that is more easily adapted to other indicators.

As an alternative to the previously discussed solution to the set covering problem, Schöbel [11] describes an approach that uses a graph to obtain the solution. Even though the algorithm is described for the weighted case, where we want to minimise the sum of the weights of the chosen elements, we can choose a weight of 1 for each element, obtaining a different algorithm for the same problem.

This algorithm assumes that the columns of the adjacency matrix have the "consecutive ones property" as well as the rows, as we assert in Proposition 5.1.1. Therefore, we define $s_i = \min(\{1 \leq j \leq n : \epsilon(b_i, b_j) \leq t_\epsilon\})$ and $e_i = \max(\{1 \leq j \leq n : \epsilon(b_i, b_j) \leq t_\epsilon\})$ as the first and last elements covered by b_i, respectively.

Proposition 5.1.1. *For a given threshold for ϵ-indicator, t_ϵ, the columns of the adjacency matrix of the corresponding graph have the "consecutive ones property" (C1P).*

Proof. Let us recall that the adjacency matrix has a value of 1 in entry (i, j) only if $\epsilon(b_j, b_i) \leq t_\epsilon$, and otherwise has a value of 0. We know that $\epsilon(b_{j+\ell}, b_j) < \epsilon(b_{j+\ell+1}, b_j)$, for $\ell > 0$ (see Proposition 5.2.1). Additionally, $\epsilon(b_j, b_j) = 1$ and $\epsilon(b_{j+1}, b_j) > 1$, since b_j and b_{j+1} are mutually non-dominated. Similarly, $\epsilon(b_{j-\ell}, b_j) < \epsilon(b_{j-\ell-1}, b_j)$, for $\ell > 0$, and $\epsilon(b_{j-1}, b_j) > 1$. Therefore, fixing a value of j, we conclude that, as i increases, $\epsilon(b_i, b_j)$ decreases when $i < j$ and increases when $i > j$. Using the notation described above, for $s_j \leq i \leq j$, $\epsilon(b_i, b_j) \leq \epsilon(b_{s_j}, b_j) \leq t_\epsilon$, since $\epsilon(b_i, b_j)$ is decreasing. Similarly, for $j < i \leq e_j$, $\epsilon(b_i, b_j) \leq \epsilon(b_{e_j}, b_j) \leq t_\epsilon$, since the sequence is increasing. Therefore, for a given j, every row $s_j \leq i \leq e_j$ has a value of 1. □

Given a threshold t_ϵ, this algorithm defines a graph G_ϵ, with the following nodes: $\{h_s, h_1, \ldots, h_n, h_t\}$. The graph has an arc (h_s, h_i) if the corresponding $b_i \in B$ covers the first element, that is, if the entry on the first row and i-th column has the value 1. In our case, this is equivalent to $\epsilon(b_i, b_1) \leq t_\epsilon$. Similarly, there is an arc (h_i, h_t) if $\epsilon(b_i, b_n) \leq t_\epsilon$.

As for the arcs between two of the middle elements, there is an arc (h_i, h_j) with $i < j$, if choosing b_i and b_j and choosing no b_ℓ, with $i < \ell < j$ does not leave any element uncovered. In other words, we add an arc from h_i to h_j with $i < j$ if $s_j \leq e_i + 1$.

Moreover, a path from h_s to h_t represents an acceptable subset for a given threshold. This is true because h_s is only connected to nodes that represent elements that cover the first element. Moreover, the structure of the graphs ensures that all elements from b_1 to b_j are covered, because of the condition for creating the arcs. Finally, only elements that cover the last element are connected to h_t, which means that every element, from b_1 to b_n is covered.

In Figure 5.1, an example is presented, with the graph G_i on the left, as presented in Section 4.1, and the corresponding graph G_ϵ on the right. The figure also illustrates the rule for the arcs in G_ϵ. For example, if there was an arc from h_2 to h_6, then choosing the respective elements, b_2 and b_6, would leave b_4 uncovered. Therefore, we only have an arc between two elements if that does not happen. Therefore, by picking a path, like the blue path, we know that its nodes represent a subset. In this case, the nodes h_2, h_5 and

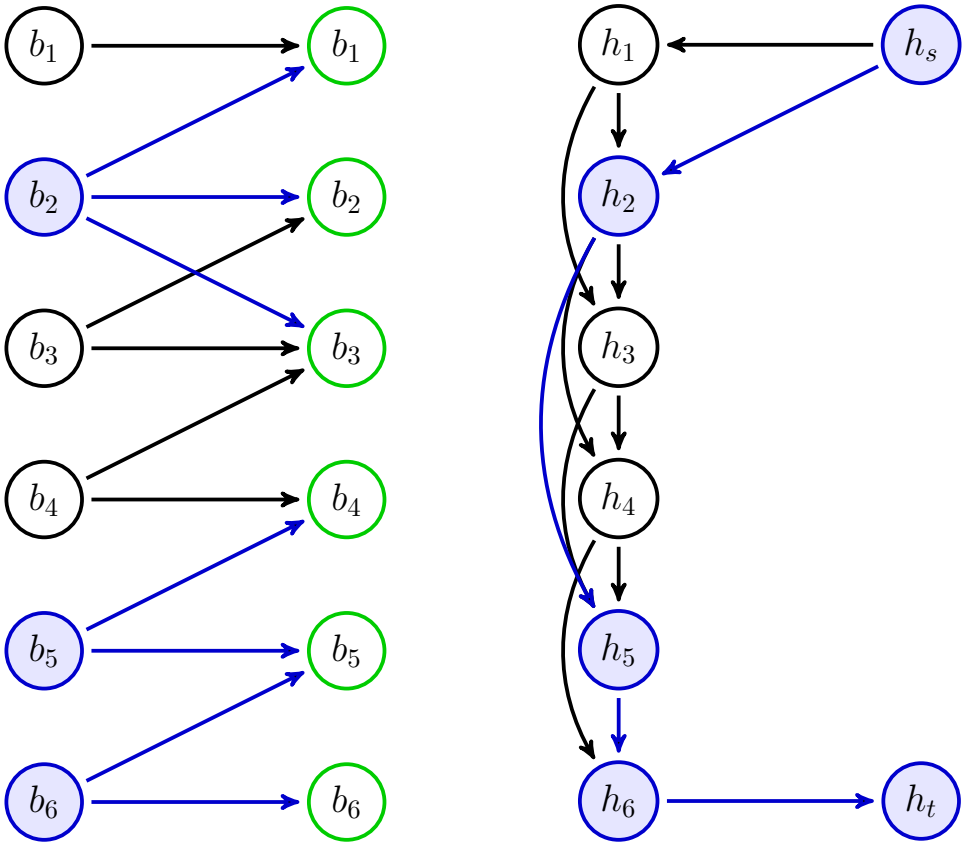

Figure 5.1: Graph G_i representing set covering (left) and respective graph G_ϵ (right)

h_6, represent the subset formed by b_2, b_5 and b_6, which is a feasible solution.

We can now find the shortest path in G_ϵ, which corresponds to the smallest subset. To do this, we visit each node, in the order $h_s, h_1, \ldots, h_n, h_t$. Then, for each node, we visit all the outgoing arcs, in order to update the size of the shortest path in the destination nodes. After visiting every node, if the shortest path from h_s to h_t goes through at most $k+1$ arcs, then there is a feasible solution, and that path represents a subset R, with $|R| \geq k$.

The complexity of this procedure is $\mathcal{O}(n^2)$, since we need to visit each arc to find the shortest path from h_s to h_t. Therefore, we now have $T_C(n) = \mathcal{O}(n^2)$ and the complexity for this version is $\mathcal{O}(n^2 \log n)$.

5.2 Dynamic Programming

We now present an algorithm using dynamic programming, similar to the one presented by Paquete et al. [8] for a different indicator. The use of dynamic programming requires the existence of a subproblem. In this case, we can consider the subproblem of finding the subset of $k-1$ elements in a subset of B. Particularly, we sort the elements of B according to the first coordinate, and denote the sorted elements by $b_1, b_2, \ldots, b_n$. Then, the considered subproblem consists in selecting $k' < k$ elements from the subset $\{b_j, \ldots, b_n\}$, with $1 \leq j \leq n$.

The main idea of the algorithm is to add an element to the subset, using the result for the subproblem in order to speed-up the calculation. Indeed, if we have a subset whose

leftmost element is b_ℓ and want to add an element b_j, with $j < \ell$, then we know that for every element b_m, with $\ell < m$, b_ℓ is preferable to b_j, since $\epsilon(b_m, b_j) > \epsilon(b_m, b_\ell)$, by Proposition 5.2.1. In other words, b_j only ϵ-dominates b_m if b_ℓ also ϵ-dominates it.

Consequently, we only need to check the elements b_m, with $j < m < \ell$. Moreover, b_ℓ will be preferable to any element to its right, by Proposition 5.2.1, which means we only need to check b_j and b_ℓ.

Formally, we denote as $R_{i,j}$ the optimal subset of cardinality i that contains b_j and is contained in $\{b_j, \ldots, b_n\}$, with the ϵ-indicator value of $T(i, j)$. The base case, $T(1, j)$, corresponds to the value obtained by considering element b_j. Since b_j must ϵ-dominate all b_m, $j < m < n$, $T(1, j)$ takes the maximum value of $\epsilon(b_j, b_m)$. Also, given that the maximum value occurs for b_n, by Proposition 5.2.2, then $T(1, j) = \epsilon(b_j, b_n)$.

The value $T(i, j)$ can be obtained by considering a subset of $i-1$ elements, with value $T(i-1, \ell)$, with $\ell > j$, and adding b_j to it. This can be expressed as:

$$T(i, j) = \min_{j<\ell\leq n-i+2} \max\left(\delta_{j,\ell}, T(i-1, \ell)\right)$$

where $\delta_{j,\ell}$ represents the correction to the ϵ-indicator value in order to consider the elements b_m, with $j < m < \ell$. As we have seen before, we only need to check the elements b_m against b_j and b_ℓ, obtaining:

$$\delta_{j,\ell} = \max_{j<m<\ell} \min\left(\epsilon(b_j, b_m), \epsilon(b_\ell, b_m)\right)$$

However, for a given j, the elements $b_1, \ldots, b_{j-1}$ are not considered in $T(k, j)$, so we need to correct the values to account for this fact. We also know that $\epsilon(b_j, b_1) > \epsilon(b_j, b_m)$, for $1 < m < j$. Therefore, when correcting the value we only need to consider the value for b_1. Consequently, the optimum value of the ϵ-indicator is given by:

$$\min_{1\leq j\leq n-k+1} \max\left(\epsilon(b_1, b_j), T(k, j)\right)$$

Therefore, the idea of the algorithm is to fill a matrix of $k \times n$ with the values of $T(i, j)$. For each entry of the matrix, we need to check all the possible values of ℓ, which are at most $\mathcal{O}(n)$. Moreover, for each value of ℓ, we need to check whether b_m is ϵ-dominated by b_j or b_ℓ, for $j < m < \ell$, in order to account for these intermediate elements. Since the values of m may also be $\mathcal{O}(n)$, the complexity for this algorithm is $\mathcal{O}(k\, n^3)$.

Moreover, the dynamic programming matrix only stores the optimal value for the ϵ-indicator. In order to obtain a subset that has that ϵ-indicator value, it is sufficient to save the next element of the subset, the value of ℓ, along with the ϵ-indicator value, for each position of the matrix. Consequently, we need one more matrix, P, defined as:

$$P(i, j) = \arg\min_{j<\ell\leq n-i+2} \max\left(\delta_{j,\ell}, T(i-1, \ell)\right)$$

Using this new matrix and given the first element, the algorithm finds the next element in the subset by looking at the values in P. Formally, given that the $(i-1)$-th element of the subset is the element with index j, then the i-th is the element with the index given by $P(n+1-i, j)$, which is simply the element with column j, taken from the i-th row from the bottom. The first element of the subset is simply the one that is chosen during the final correction of the ϵ-indicator value, and is given by:

$$\arg\min_{1\leq j\leq n-k+1} \max\left(\epsilon(b_1, b_j), T(k, j)\right)$$

Proposition 5.2.1. *For $a, a', b, b' \in B$, with $b'_1 < a_1 < a'_1 < b_1$, i) $\epsilon(a, b) > \epsilon(a', b)$ and ii) $\epsilon(a, b') < \epsilon(a', b')$*

Proof. We start by proving that $\epsilon(a, b) > \epsilon(a', b)$, that is,

$$\max\left(\frac{b_1}{a_1}, \frac{b_2}{a_2}\right) > \max\left(\frac{b_1}{a'_1}, \frac{b_2}{a'_2}\right)$$

We know that $a_1 < a'_1 < b_1$ and $a_2 > a'_2 > b_2$. This means that $b_1/a_1 > b_2/a_2$, and equivalently for a', so we have,

$$\frac{b_1}{a_1} > \frac{b_1}{a'_1}$$

However, this is equivalent, by manipulation, to $a_1 < a'_1$, which we know to be true.

Similarly, we know that $b'_1 < a_1 < a'_1$ and $b'_2 > a_2 > a'_2$. Consequently, $b'_1/a_1 < b'_2/a_2$, and equivalently for a', so the inequality is equivalent to

$$\frac{b'_2}{a_2} < \frac{b'_2}{a'_2}$$

However, this is equivalent, by manipulation, to $a_2 > a'_2$, which we know to be true. □

Proposition 5.2.2. *For $a, a', b, b' \in B$, with $a_1 < b_1 < b'_1 < a'_1$, i) $\epsilon(a, b) < \epsilon(a, b')$ and ii) $\epsilon(a', b) > \epsilon(a', b')$.*

Proof. We start by proving that $\epsilon(a, b) < \epsilon(a, b')$, that is,

$$\max\left(\frac{b_1}{a_1}, \frac{b_2}{a_2}\right) < \max\left(\frac{b'_1}{a_1}, \frac{b'_2}{a_2}\right)$$

We know that $a_1 < b_1 < b'_1$ and $a_2 > b_2 > b'_2$. This means that $b'_1/a_1 > b'_2/a_2$, and equivalently for b and a, so we have,

$$\frac{b_1}{a_1} < \frac{b'_1}{a_1}$$

This is equivalent to having $b_1 < b'_1$, which is true.

As for proving that $\epsilon(a', b) > \epsilon(a', b')$, we know that $b_1 < b'_1 < a'_1$ and $b_2 > b'_2 > a'_2$. Consequently, $b'_1/a'_1 < b'_2/a'_2$, and equivalently for b and a', so we have,

$$\frac{b_2}{a'_2} > \frac{b'_2}{a'_2}$$

This is equivalent to having $b_2 > b'_2$, which we know to be true. □

5.2.1 Improvements

m-improvement

The calculation of $\delta_{j,\ell}$ requires an extra cycle, and since we have $j < m < \ell$, m can take $\mathcal{O}(n)$ different values. However, not only does this calculation allow a more efficient implementation, but it can also be executed only once, without repeating it for each i, by pre-calculating the values.

The main idea is that, given Propositions 5.2.2, $\epsilon(b_j, b_m)$ increases and $\epsilon(b_\ell, b_m)$ decreases, as m increases. Since the value of $\delta_{j,\ell}$ is the maximum over the values of m, we can easily calculate it by knowing the right value of m, which we denote by m'_ℓ. By Proposition 5.2.1, $\epsilon(b_\ell, b_m) < \epsilon(b_{\ell+1}, b_m)$, which means that $m'_\ell \leq m'_{\ell+1}$.

Knowing this, it is more efficient to simply increase the value of m between values of ℓ, which allows us to pre-calculate all the values for a given m while increasing the value of m at most $\mathcal{O}(n)$ times when keeping j constant. Therefore, this procedure has amortised time complexity $\mathcal{O}(n^2)$, and the algorithm is now $\mathcal{O}(k\,n^2)$, since we simply need to lookup the value of $\delta_{j,\ell}$.

ℓ-improvement

Using the improvement described above, the values $\delta_{j,\ell}$ are pre-calculated in $\mathcal{O}(n^2)$ time. Since $\mathcal{O}(n^2)$ is also the number of values we calculate, this complexity is optimal, unless we choose to calculate a smaller number of values.

On the other hand, the dynamic programming matrix, T takes $\mathcal{O}(k\,n^2)$ complexity to calculate, even though there are only $\mathcal{O}(k\,n)$ values. Note that the algorithm iterates over the values of ℓ, the next element in the subset, in order to find the best possibility.

Wang et al. [12] suggest an improvement for a related problem, which can be adapted for this algorithm. This improvement removes the need to iterate over values of ℓ, as it is possible to test a small number of these values, given the value used previously. Formally, knowing that $P(i,j) \leq P(i,j+1)$, and given that there are at most n different values for $P(i,j)$, we expect the difference between $P(i,j)$ and $P(i,j+1)$ to be around one, on average.

Moreover, it is important to know that the value for the ϵ-indicator considering that $\ell = P(i,j)$ is the only local minimum. This means that if by increasing or decreasing ℓ, the value of ϵ-indicator increases, we know that we have found the minimum, and so $P(i,j) = \ell$.

We will now demonstrate that $T(i,j)$ and $\delta_{j,\ell}$ are non-increasing, when j increases, and $\delta_{j,\ell}$ is non-decreasing, as ℓ increases.

Proposition 5.2.3. *For $j < \ell$, $\delta_{j,\ell} \geq \delta_{j+1,\ell}$*

Proof. We know that:

$$\delta_{j,\ell} = \max_{j<m<\ell} \min\left(\epsilon(b_j, b_m), \epsilon(b_\ell, b_m)\right)$$

For $j < j+1 < m$, we know, by Proposition 5.2.1, that $\epsilon(b_j, b_m) > \epsilon(b_{j+1}, b_m)$. Consequently, $\min\left(\epsilon(b_{j+1}, b_m), \epsilon(b_\ell, b_m)\right) \leq \min\left(\epsilon(b_j, b_m), \epsilon(b_\ell, b_m)\right)$, and:

$$\delta_{j+1,\ell} = \max_{j+1<m<\ell} \min\left(\epsilon(b_{j+1}, b_m), \epsilon(b_\ell, b_m)\right) \leq \max_{j+1<m<\ell} \min\left(\epsilon(b_j, b_m), \epsilon(b_\ell, b_m)\right)$$

At this point, we know that adding the possibility $m = j + 1$ to the maximum will not decrease the value, and therefore:

$$\delta_{j+1,\ell} \leq \max_{j+1<m<\ell} \min\left(\epsilon(b_j, b_m), \epsilon(b_\ell, b_m)\right) \leq \max_{j<m<\ell} \min\left(\epsilon(b_j, b_m), \epsilon(b_\ell, b_m)\right) = \delta_{j,\ell}$$

□

Proposition 5.2.4. *For $j < \ell$, $\delta_{j,\ell} \leq \delta_{j,\ell+1}$*

Proof. We know that:

$$\delta_{j,\ell} = \max_{j<m<\ell} \min\left(\epsilon(b_j, b_m), \epsilon(b_\ell, b_m)\right)$$

For $m < \ell < \ell + 1$, we know, by Proposition 5.2.1, that $\epsilon(b_\ell, b_m) < \epsilon(b_{\ell+1}, b_m)$. Consequently, $\min\left(\epsilon(b_j, b_m), \epsilon(b_\ell, b_m)\right) \leq \min\left(\epsilon(b_j, b_m), \epsilon(b_{\ell+1}, b_m)\right)$, and:

$$\delta_{j,\ell} = \max_{j<m<\ell} \min\left(\epsilon(b_j, b_m), \epsilon(b_\ell, b_m)\right) \leq \max_{j<m<\ell} \min\left(\epsilon(b_j, b_m), \epsilon(b_{\ell+1}, b_m)\right)$$

At this point, we know that adding the possibility $m = \ell$ to the maximum will not decrease the value, and therefore:

$$\delta_{j,\ell} \leq \max_{j<m<\ell} \min\left(\epsilon(b_j, b_m), \epsilon(b_{\ell+1}, b_m)\right) \leq \max_{j<m<\ell+1} \min\left(\epsilon(b_j, b_m), \epsilon(b_{\ell+1}, b_m)\right) = \delta_{j,\ell+1}$$

□

Proposition 5.2.5. *For $1 \leq j \leq n - i$, $T(i, j) \geq T(i, j + 1)$*

Proof. We know that $I_\epsilon(R_{i,j}, \{b_j, \ldots, b_n\}) = T(i, j)$.

We denote $B_{i,\ldots,j} = \{b_i, \ldots, b_j\}$ and by R' the set obtained by replacing b_j for b_{j+1} in the set $R_{i,j}$, that is, $R' = (R_{i,j} \cap B_{j+1,\ldots,n}) \cup \{b_{j+1}\}$. Given that $T(i, j + 1)$ is the optimal value of the ϵ-indicator for the subset $B_{j+1,\ldots,n}$, then $T(i, j + 1) \leq I_\epsilon\left(R', \{b_{j+1}, \ldots, b_n\}\right)$. By Proposition 5.2.1, $\epsilon(b_j, b_m) \geq \epsilon(b_{j+1}, b_m)$, for $m > j + 1$. Therefore, using the fact that R and R' only differ by one element (b_{j+1} instead of b_j), and that b_{j+1} results in a lower ϵ-indicator value, we have that:

$$\begin{aligned}
T(i, j+1) &\leq I_\epsilon\left(R', B_{j+1,\ldots,n}\right) \\
&\leq \max_{b \in B_{j+1,\ldots,n}} \min_{r \in R'} \epsilon(r, b) \\
&\leq \max_{b \in B_{j+1,\ldots,n}} \min_{r \in R} \epsilon(r, b) \\
&\leq \max_{b \in B_{j,\ldots,n}} \min_{r \in R} \epsilon(r, b) \\
&\leq I_\epsilon\left(R, B_{j,\ldots,n}\right) \\
&\leq T(i, j)
\end{aligned}$$

□

The value of $T(i, j)$ is calculated as the minimum over values of ℓ of the maximum of one non-increasing sequence ($T(i, \ell)$) and one non-decreasing sequence ($\delta_{j,\ell}$), as demonstrated by Propositions 5.2.5 and 5.2.4, respectively. Therefore, the maximum of the two

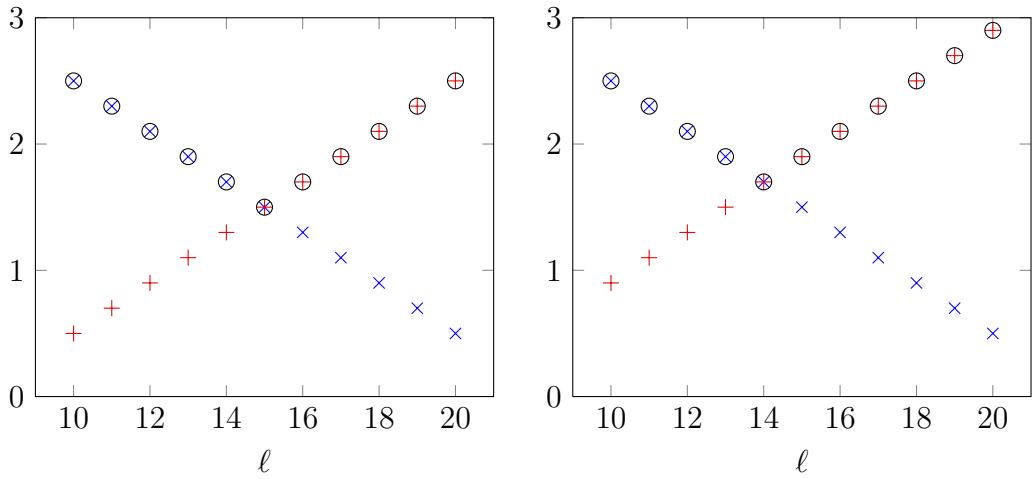

Figure 5.2: Illustration of non-increasing and non-decreasing sequences and their maximum; see text for more details

sequences will be a bitonic sequence which decreases until the non-increasing sequence becomes smaller, and then increases. Consequently, there is only one minimum, close to where the non-increasing sequence becomes smaller. This is represented in the plots of Figure 5.2, with $T(i,\ell)$ (in blue) decreasing until $\delta_{j,\ell}$ (in red) becomes larger.

Moreover, if we now look at the calculation of $T(i,j-1)$, the non-increasing sequence remains equal, but by Proposition 5.2.3, $\delta_{j-1,\ell} \geq \delta_{j,\ell}$, which means that the non-decreasing sequence may increase for every ℓ. Consequently, the non-decreasing sequence may overcome the non-decreasing sequence for a smaller value of ℓ, which means that $P(i,j-1) \leq P(i,j)$.

Figure 5.2 illustrates this situation, with $\delta_{j,\ell}$ represented with red plus marks, $T(i,\ell)$ with blue crosses and the maximum of the two sequences with black circles. In the left plot, we have the sequences for a certain j, whereas in the right plot we have the same sequences for $j-1$. Since $\delta_{j-1,\ell} \geq \delta_{j,\ell}$, the minimum of the maximum of the two subsequences moves to the left in the right plot. This position is represented by the ℓ value of the lowest black circle, which is at $\ell = 15$ in the left plot and $\ell = 14$ in the right plot. The index of this minimum is stored in P, and consequently, this figure illustrates that $P(i,j-1) \leq P(i,j)$.

In conclusion, instead of iterating over all the possible values of ℓ, the algorithm stops when the next value of ℓ increases the ϵ-indicator. Also, the value of $P(i,j)$ for a given value of j is stored and used as an upper bound for $P(i,j-1)$. The complexity for calculating all the values $T(i,j)$ is now $\mathcal{O}(k\,n)$ amortised, since on average, the algorithm checks $\mathcal{O}(1)$ values of ℓ. Adding the preprocessing in $\mathcal{O}(n^2)$ and considering that $k \leq n$, this version of the algorithm has $\mathcal{O}(n^2)$ time complexity.

5.3 Complexity and Results

We now have four distinct versions with the complexities given below:

- Threshold Algorithm $\mathcal{O}(n^2 \log n)$
- Dynamic Programming $\mathcal{O}(k\,n^3)$

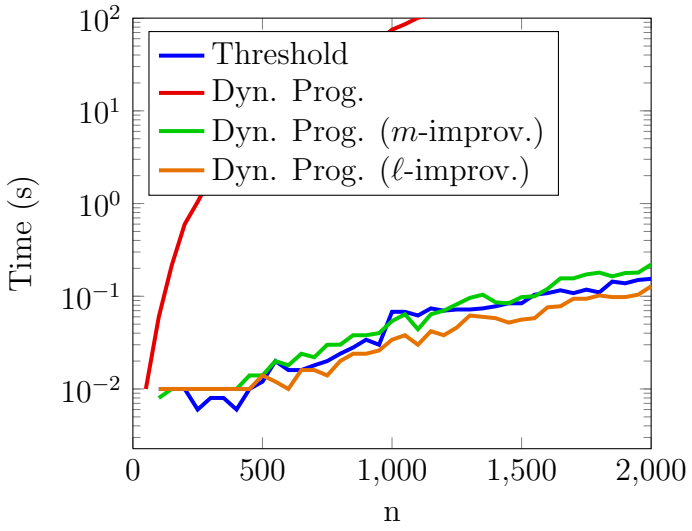

Figure 5.3: Running time for the four approaches with $50 \leq n \leq 2000$, $k = 20$

- Dynamic Programming and m-improvement $\mathcal{O}(k\,n^2)$
- Dynamic Programming and ℓ-improvement $\mathcal{O}(n^2)$

The Dynamic Programming algorithms get progressively better with each improvement, with the best complexity being $\mathcal{O}(n^2)$. Unlike the first two Dynamic Programming versions, the time complexity of this version does not depend on k, similarly to the Threshold Algorithm.

5.3.1 Experimental Results

Two tests were run, in order to compare the four versions of the algorithm and the influence of the parameters n and k on the running time. We refer to Chapter 3 for a more detailed discussion of the the experimental setup.

The results for the four approaches are present in Figure 5.3, considering k is constant. The naive Dynamic Programming version is very slow in practice, which is natural if we consider it has $\mathcal{O}(k\,n^3)$ complexity. Even when the value of n is around 200, this algorithm already takes more time to finish that any of the others for $n = 2000$.

Regarding the remaining versions, the running times seem to be consistent with the complexities. Perhaps the largest surprise is the small difference between the Threshold Algorithm and the first version of the Dynamic Programming algorithm. However, since k takes a relatively small constant value in this test, the complexity of this version of the dynamic programming algorithm becomes $\mathcal{O}(n^2)$.

On the other hand, in Figure 5.4 we can see what happens to the running time when we keep n constant and vary k. Once more, the difference between the naive version of the Dynamic Programming algorithm and the remaining three is clear. The naive version does not even appear in the graph, since it takes too long to complete even the smaller case ($n = 2000$, $k = 50$), taking 1383.11 seconds.

The relevance of the factor k in the complexity also becomes clear, since the version with complexity $\mathcal{O}(k\,n^2)$ is very affected by the increase in the value of k. In fact, if we consider $k = 1000$, the version with m-improvement takes around twenty times more

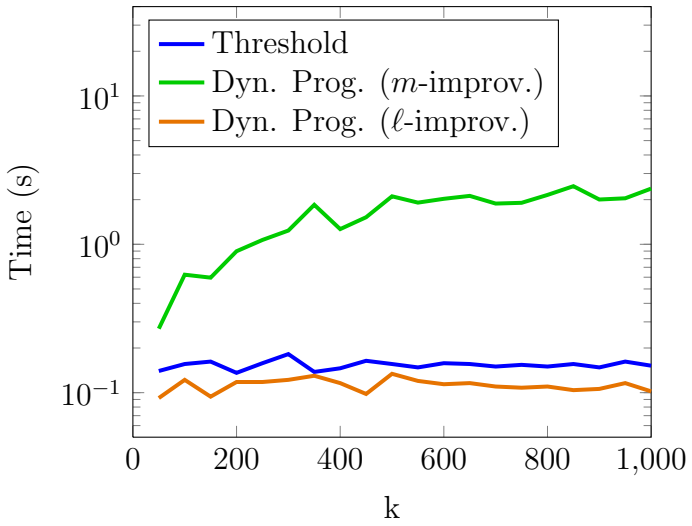

Figure 5.4: Running time for the four approaches with $n = 2000$, $50 \leq k \leq 1000$

time than the other versions.

Finally, comparing the remaining versions, it seems clear that the Dynamic Programming algorithm consistently performs better than the Threshold Algorithm, even if it is a small difference, which is consistent with the time complexities of both algorithms.

Chapter 6

Representation Problem using Coverage

In this chapter, we consider the same representation problem as in Chapter 5, using the Coverage indicator, introduced by Sayin [10], instead of the ϵ-indicator. As before, we want to find a subset with a given cardinality that minimises a given property, but in this case the property is related to the distance between the selected elements and the elements of the set. This problem is related to centroid-based clustering, where the chosen elements represent the cluster centers, and we want to minimise the maximum distance of an element to the closest center.

Formally, given a set of vectors B and k, this problem consists in finding a subset $R \subseteq B$, with $|R| = k$ with the minimum value of the Coverage indicator, that is, for each element of B there is an element in R for which their distance is less than the value of Coverage. Using the notation introduced in Chapter 2, we have the goal given by:

$$\arg\min_{\substack{R \subseteq B \\ |R|=k}} I_C(R, B)$$

where I_C represents the Coverage indicator, introduced in Chapter 1 and given by:

$$I_C(R, B) = \max_{b \in B} \min_{r \in R} ||r - b||$$

and $||a - b||$ represents any p-norm between of the vector $a - b$.

In this chapter we prove that the same algorithms that are presented in Chapter 5 may be used for Coverage, by simply changing the indicator value. Additionally, since the algorithms are almost similar, with the only change being the function that calculates the indicator, the experimental results are very similar to those that were analysed in Chapter 5.

6.1 Threshold Algorithm

The Threshold Algorithm is used for finding the optimal value of ϵ-indicator based on the reduction to a graph covering problem. By considering the distance between the points as the measure to determine which edges are present in the graph, we can use the same algorithm to find the optimal subset according to Coverage. As in the ϵ-indicator

case, we need to prove that the rows of the adjacency matrix, for the defined graphs, have the "consecutive ones property".

Proposition 6.1.1. *Given a threshold for Coverage, t_C, the rows of the adjacency matrix of its corresponding graph have the "consecutive ones property" (C1P).*

Proof. Formally, for any non-dominated elements $b, x, y, z \in B$, with $x_1 < y_1 < z_1$, we want to show that if $||b - x|| \leq t_C$ and $||b - z|| \leq t_C$, then $||b - y|| \leq t_C$.

We need two consider two cases: i) $b_1 \leq y_1$; ii) $b_1 > y_1$. For case i), we have that $b_1 < y_1 < z_1$ and $b_2 > y_2 > z_2$ (or the elements would be dominated). Therefore, we have $y_1 < z_1 \Rightarrow y_1 - b_1 < z_1 - b_1 \Rightarrow |y_1 - b_1|^p < |z_1 - b_1|^p$, since $y_1 - b_1$ and $z_1 - b_1$ are both positive. Similarly, $y_2 > z_2 \Rightarrow b_2 - y_2 < b_2 - z_2 \Rightarrow |b_2 - y_2|^p < |b_2 - z_2|^p$, since $b_2 - y_2$ and $b_2 - z_2$ are both positive. Putting both results together, we have that $|y_1 - b_1|^p + |b_2 - y_2|^p < |z_1 - b_1|^p + |b_2 - z_2|^p$, or equivalently, $||b - y|| < ||b - z||$, which implies that $||b - y|| \leq t_C$.

For the ∞-norm, $||a-b||_\infty = \max(|a_1-b_1|, |a_2-b_2|)$, it is enough that $|y_1-b_1| < |z_1-b_1|$ and $|y_2 - b_2| < |z_2 - b_2|$, which is a consequence of the case $p = 1$.

For the second case, a similar reasoning applies, using x instead of z. Therefore, we conclude that $||b - y|| \leq t_C$ for every case, thus we can conclude that the rows of the adjacency matrix have the C1P. □

The alternative set covering procedure, discussed in Section 5.1.1, is also applicable when using Coverage. As we mentioned, that algorithm is based on the "consecutive ones property" of the columns of the adjacency matrix, as well as the rows. Therefore, proving that the algorithm may be used for the problem described in this chapter is a matter of proving that using Coverage preserves the desired property, as is asserted in Proposition 6.1.2.

Proposition 6.1.2. *For a given threshold for Coverage, t_C, the columns of the adjacency matrix of the corresponding graph have the "consecutive ones property" (C1P).*

Proof. We recall that the adjacency matrix has a value of 1 in entry (i, j) if $||b_j - b_i|| \leq t_C$, and a value of 0 otherwise. We know that $||b_{j+\ell} - b_j|| < ||b_{j+\ell+1} - b_j||$, for $\ell > 0$ (see Proposition 6.2.1). This inequality is also trivially true for $\ell = 0$. Similarly, we have that $||b_{j-\ell} - b_j|| < ||b_{j-\ell-1} - b_j||$, for $\ell \geq 0$ (see Proposition 6.2.2). Therefore, by fixing a value of j, we conclude that, as i increases, $||b_i - b_j||$ decreases when $i < j$ and increases when $i > j$.

We denote, for the Coverage indicator, $s_i = \min(\{1 \leq j \leq n : ||b_i - b_j|| \leq t_C\})$ and $e_i = \max(\{1 \leq j \leq n : ||b_i - b_j|| \leq t_C\})$. Then, for $s_j \leq i \leq j$, $||b_i - b_j|| \leq ||b_{s_j} - b_j|| \leq t_C$, since $||b_i - b_j||$ is decreasing. Similarly, for $j < i \leq e_j$, $||b_i - b_j|| \leq ||b_{e_j} - b_j|| \leq t_C$, since the sequence is increasing. Therefore, for a given j, every row $s_j \leq i \leq e_j$ has a value of 1. □

6.2 Dynamic Programming

The Dynamic Programming algorithm proposed in Section 5.2 finds the minimum value for the ϵ-indicator. However, this algorithm also works for Coverage, and it is sufficient to prove that the propositions presented in that section are also true for Coverage.

Moreover, since the Coverage indicator is commutative, the first part of Proposition 5.2.1 and the second part of Proposition 5.2.2 are equivalent, as are the second part of Proposition 5.2.1 and the first part of Proposition 5.2.2.

This Dynamic Programming approach is similar to that used by Paquete et al. [8], even though the version described in this thesis is further improved.

Proposition 6.2.1. *For $a, b, c \in B$, with $a_1 < b_1 < c_1$, we have $||a - b|| < ||a - c||$.*

Proof. Since the elements are non-dominated, we have that $a_2 > b_2 > c_2$. Therefore, $b_1 - a_1 < c_1 - a_1$, and since $b_1 - a_1 > 0$ and $c_1 - a_1 > 0$, $|b_1 - a_1|^p < |c_1 - a_1|^p$. Similarly for the second coordinate, $a_2 - b_2 < a_2 - c_2$, and since both terms are positive $|a_2 - b_2|^p < |a_2 - c_2|^p$.

Concluding, $||a - b|| = |b_1 - a_1|^p + |a_2 - b_2|^p < |c_1 - a_1|^p + |a_2 - c_2|^p = ||a - c||$.For the ∞-norm, since $|c_1 - a_1| > |b_1 - a_1|$ and $|a_2 - c_2| > |a_2 - b_2|$, we conclude that $||a - c|| = \max(|c_1 - a_1|, |c_2 - a_2|) > \max(|b_1 - a_1|, |b_2 - a_2|) = ||b - a||$. □

Proposition 6.2.2. *For $a, b, c \in B$, with $a_1 < b_1 < c_1$, we have $||a - c|| > ||b - c||$.*

Proof. Since the elements are non-dominated, we have that $a_2 > b_2 > c_2$. Therefore, $c_1 - a_1 > c_1 - b_1$, and since both sides of the inequality are positive, $|c_1 - a_1|^p > |c_1 - b_1|^p$. For the the second coordinate, $a_2 - c_2 > b_2 - c_2$, and since both terms are positive $|a_2 - c_2|^p > |b_2 - c_2|^p$.

Concluding, $||a - c|| = |c_1 - a_1|^p + |a_2 - c_2|^p > |c_1 - b_1|^p + |b_2 - c_2|^p = ||b - c||$. For the ∞-norm, since $|c_1 - a_1| > |b_1 - a_1|$ and $|a_2 - c_2| > |a_2 - b_2|$, we conclude that $||a - c|| = \max(|c_1 - a_1|, |c_2 - a_2|) > \max(|c_1 - b_1|, |c_2 - b_2|) = ||b - c||$. □

Since these propositions are the only place in the proofs where the expression of $\epsilon(a, b)$ is used, it can be replaced by $||a - b||$, proving that the Dynamic Programming algorithm is also applicable to Coverage. For example, the expression for $\delta_{j,\ell}$ becomes:

$$\delta_{j,\ell} = \max_{j<m<\ell} \min\left(||b_j - b_m||, ||b_\ell - b_m||\right)$$

6.3 Complexity and Results

Given that the algorithms are the same as those used for the ϵ-indicator and that the complexity of calculating the distance is constant, as with the $\epsilon(a, b)$ function, the complexity of the algorithms is also the same, with a time complexity of $\mathcal{O}(n^2 \log n)$ for the Threshold Algorithm and $\mathcal{O}(n^2)$ amortised for the Dynamic Programming algorithm.

Given that the same algorithms (and same implementation) are used for both indicators, the experimental results are similar to the ones obtained for the ϵ-indicator. We refer to Section 5.3 for a discussion of these results.

Chapter 7

Representation Problem using Uniformity

In this chapter, we use a third indicator, Uniformity, to select the representative subset. This indicator is different than the ones previously seen, since it measures how far apart the points in the chosen subset are, instead of comparing in some way the chosen subset with the original set, as the ϵ-indicator and Coverage.

Uniformity was introduced by Sayin [10], and is formally defined as the minimum distance between any pair of elements in the subset, that is:

$$I_U(R) = \min_{\substack{i,j \in \{1,\dots,|R|\} \\ i \neq j}} ||p_i - p_j||$$

.

The objective of this problem is to find a subset of a given cardinality such that the minimum distance between any pair of elements is maximised. Using the notation introduced in Chapter 2, we want to find:

$$\arg\max_{\substack{R \subseteq B \\ |R|=k}} I_U(R, B)$$

In this chapter we present two algorithms that solve the representation problem using Uniformity, based on the Threshold Algorithm and Dynamic Programming approach presented in Chapter 5. The goal of this Chapter is to present the algorithms that are then combined into the multiobjective algorithms presented in the following chapters.

7.1 Threshold Algorithm

The Threshold Algorithm for Uniformity uses the same idea that is described in Chapter 4 and Section 5.1. The Uniformity values are distances between the elements, and consequently the algorithm sorts the distance values between the pairs of elements, and does a binary search over these values.

Therefore, the only necessary change is on the set covering procedure, which needs to be changed to check if for a given Uniformity threshold, a solution exists. This modification is not trivial, since Uniformity measures distances between chosen elements, unlike Coverage or ϵ-indicator, which are measured between the chosen elements and entire set.

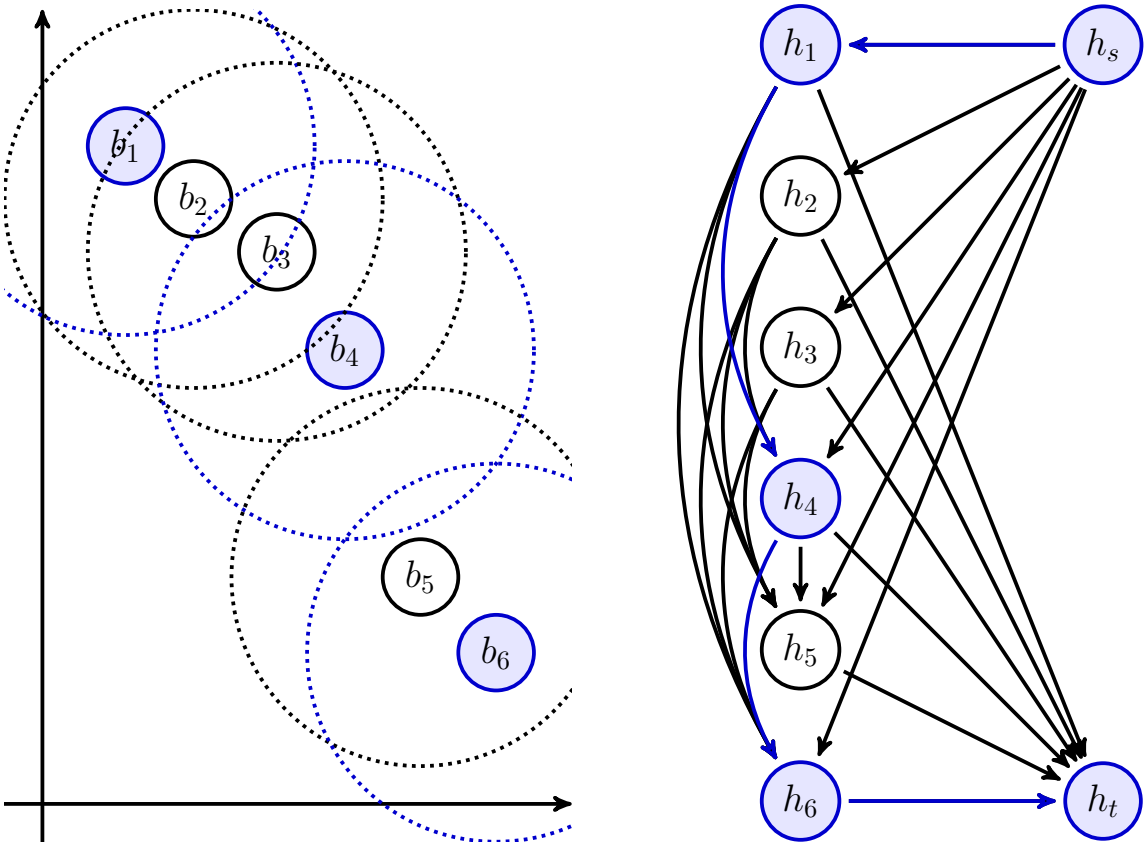

Figure 7.1: Set of 2D points (left) and respective graph G_U (right)

Setting a threshold for Uniformity means that we want to find a subset with at least k elements such that the distances between those elements is greater or equal to the threshold. Even though we want a subset with k elements, we may remove any elements without the Uniformity decreasing, since it is calculated as the minimum of all the possible distances, and by removing an element, we simply remove some distance values.

Given a threshold value t_U, we define a directed acyclic graph G_U with $n+2$ vertices: a source node h_s, a target node h_t and a node h_i corresponding to each element of $b_i \in B$. The graph G_U has arcs (h_s, h_i) and (h_i, h_t) for every $1 \leq i \leq n$. The graph G_U also has an arc (h_i, h_j) for each pair of nodes such that $i < j$ and $||b_i - b_j|| \geq t_U$.

An example is given in Figure 7.1, where we have a plot of the points, on the left, with the Uniformity threshold identified with a dotted circumference. The corresponding graph is also presented on the right, with an arc between two nodes only of their distance is greater than the threshold.

We now argue that any path from h_s to h_t on graph G_U corresponds to a subset whose value of Uniformity is at least t_U, that is, if there is a path $P = (h_s, h_{i_1}, \ldots, h_{i_m}, h_t)$, then $I_U(\{b_{i_1}, \ldots, b_{i_m}\}) \geq t_U$. This is true because the elements b_i are ordered by the first coordinate. Therefore, since there is an arc $(h_{i_j}, h_{i_{j+1}})$ in the graph, then $t_U \leq ||b_{i_j} - b_{i_{j+1}}||$ and by Proposition 6.2.1, $t_U \leq ||b_{i_j} - b_{i_{j+1}}|| \leq ||b_{i_j} - b_\ell||$ for $\ell > i_{j+1}$. In conclusion, the distance between the elements of B corresponding to any two nodes of a path is always at least t_U.

Therefore, there is an feasible subset for a given t_U if and only if there is a path with at least $k+2$ nodes or $k+1$ arcs in the corresponding graph G_U (including h_s and

h_t). In Figure 7.1, one possible subset is presented, consisting of the blue nodes. The corresponding path is also marked in blue, which as explained passes through h_s, each of the selected elements, and then h_t.

This problem is reduced to finding the maximum path in G_U. In order to do this, the algorithm builds the graph as described, and then visits each node, in the order $h_s, h_1, \ldots, h_n, h_t$, storing the size of the longest path from h_s. When a node is visited, all the outgoing arcs from that node are also visited, and the destination nodes are updated. After visiting every node, if the longest path from h_s to h_t goes through at least $k+1$ arcs, then there is a feasible solution, since that path represents a subset R, with $|R| \geq k$. Since the Uniformity is calculated as the minimum of all the pairs of distances, we may remove elements from the subset until we reach the desired cardinality, since removing points does not decrease the Uniformity value.

Since we need to visit each edge once, to correctly calculate the length of the longest paths to each node, this procedure has complexity $\mathcal{O}(n^2)$. Using the notation analogous to the one defined in Section 4.1, we now have $T_C(n) = \mathcal{O}(n^2)$. Therefore, the complexity of the Threshold Algorithm for Uniformity is $\mathcal{O}(n^2 \log n)$.

7.2 Dynamic Programming

The Dynamic Programming algorithm for Uniformity is similar to the one described in Section 5.2, and is similar to the approach described by Wang and Kuo [12]. This algorithm is based on adding an element to a subset of size $i-1$ to obtain a subset of size i. Let $S_{i-1,\ell} \subseteq \{b_\ell, \ldots, b_n\}$, with $b_\ell \in S_{i-1,\ell}$ and $|S_{i-1,\ell}| = i-1$ and b_j the element to add to that subset, with $j < \ell$. Since the Uniformity indicator calculates the minimum over all pairs of elements,

$$I_U(\{b_j\} \cup S_{i-1,\ell}) = \min\left(\min(\{||b_j - e|| : e \in S_{i-1,\ell}\}), I_U(S_{i-1,\ell})\right)$$

Additionally, by Proposition 6.2.1, $||b_j - b_\ell|| < ||b_j - e||$, for $e \in S_{i-1,\ell} \setminus \{b_\ell\}$, so we can simplify I_U as:

$$I_U(\{b_j\} \cup S_{i-1,\ell}) = \min\left(||b_j - b_\ell||, I_U(S_{i-1,\ell})\right)$$

This shows that we can calculate the Uniformity indicator incrementally by calculating the distances between the two first elements of the new set, and consequently we can calculate $T(i,j)$ using the following expression:

$$T(i,j) = \max_{j<\ell\leq n-i+2} \min\left(||b_j - b_\ell||, T(i-1,\ell)\right)$$

Unlike Coverage and ϵ-indicator, for which we need to do a final correction, the Uniformity indicator does not depend on the unchosen elements, and therefore the results present in the last row represent feasible solutions. Therefore, the final solution is given by:

$$\min_{1\leq j\leq n-k+1} T(k,j)$$

The time complexity for this algorithm is the same as the algorithm described in Section 5.2, since the algorithms are similar. Additionally, an improvement similar to that

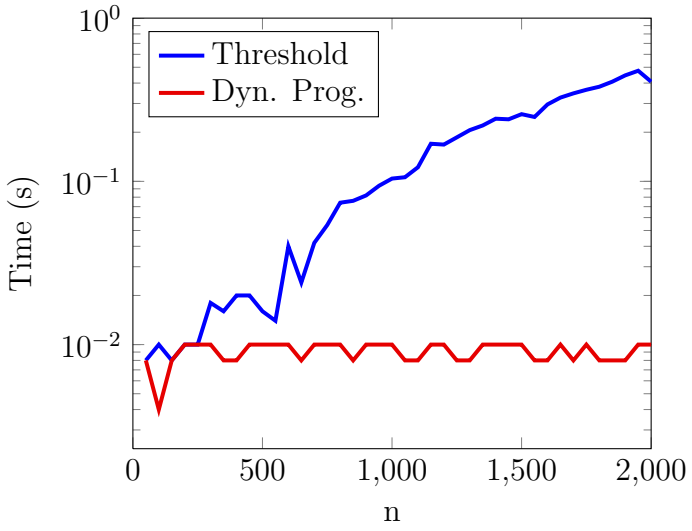

Figure 7.2: Running time for the four approaches with $50 \leq n \leq 2000$, $k = 20$

of Section 5.2.1 can be applied, as explained by Wang et al. [12], yielding the complexity of $\mathcal{O}(k\,n)$.

7.3 Complexity and Results

In this chapter, we present two different approaches to solve the discussed problem, whose complexities are:

- Threshold Algorithm $\mathcal{O}(n^2 \log n)$
- Dynamic Programming $\mathcal{O}(k\,n)$

7.3.1 Experimental Results

We run two tests with the two approaches, in order to study the influence of the size of the set, n, and the subset k, in the performance of the algorithms. The methodology of the experiments is discussed in detail in Chapter 3.

In Figure 7.2, the results for the first test are presented. As expected, the Threshold Algorithm implementation grows quadratically with n, and the difference to the Dynamic Programming version increases to 10 times more. Considering that the Dynamic Programming version depends linearly of n, it is not surprising that it has much better performance than the Threshold Algorithm.

On the second test, we study the influence of k. Since the complexity of the Threshold Algorithm is not dependent on k, it is expected that the algorithm has constant performance. However, the Dynamic Programming approach depends on the size of the subset, and consequently may take more time to solve instances with a larger value of k.

The results for this test are presented in Figure 7.3, using a logarithmic scale for the running time. The results indicate that, even though the Threshold Algorithm has a similar running time for different values of k, and that the performance of the Dynamic Programming approach decreases as k increases, it is still better by an order of magnitude.

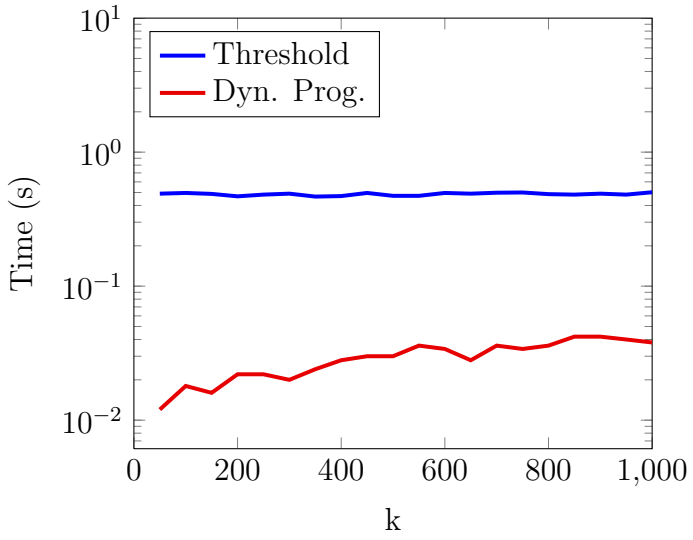

Figure 7.3: Running time for the four approaches with $n = 2000$, $50 \leq k \leq 1000$

Additionally, since k is the size of the subset, k is at most n, since it does not make sense to extract more than n elements. Therefore, the complexity of the Dynamic Programming approach, $\mathcal{O}(k\,n)$, is in fact at most $\mathcal{O}(n^2)$, which is still better than the complexity of the Threshold Algorithm. Therefore, even considering $k = n$, the complexity of the Dynamic Programming algorithm would be better, and consequently it is not surprising that it outperforms the Threshold Algorithm.

Chapter 8

Representation Problem using ϵ-indicator and Coverage

We now expand the representation problem to consider more than one indicator. As before, we want to find a subset with a given cardinality, but this time we want to optimise two different indicators, specifically Coverage and the ϵ-indicator. Therefore, we now have a multiobjective version of the representation problem.

By optimising two indicators, we obtain not only the best solutions for each indicator, but compromise solutions as well. Therefore, considering the multiobjective problem allows us to obtain different solutions, and these solutions may present a different representation of the problem, which may be interesting for the decision maker.

Formally, given a set of vectors B and a cardinality k, we want to find subsets $R \subseteq B$, with $|R| = k$, such that the pair of values of Coverage and ϵ-indicator are non-dominated. These subsets R must ϵ-dominate B and satisfy the condition that for each element of B there is an element in R for which their distance is less than the value of Coverage. Using the notation introduced in Chapter 2, we have the goal given by:

$$\underset{\substack{R \subseteq B \\ |R|=k}}{\arg\min} \; (I_\epsilon(R, B), I_C(R, B))$$

Here, the arg min operator returns the non-dominated solutions, that is, given the pairs of acceptable ϵ-indicator and Coverage values, it interprets them as 2D vectors, and extracts the non-dominated set.

In this chapter, we present two algorithms that solve this problem, based on the Threshold Algorithm presented in Chapters 4 and 5 and the Dynamic Programming approach presented in Chapter 5. Particularly, we discuss the changes needed to adapt the algorithms to a multiobjective problem, and also the modifications needed to introduce the Coverage algorithm.

8.1 Threshold Algorithm

The Threshold Algorithm is used to find the optimal value of ϵ-indicator, and is, therefore, designed to search for the best value on a one dimensional space. For this problem, we need to adapt the algorithm to search in the 2D space, and keep the non-

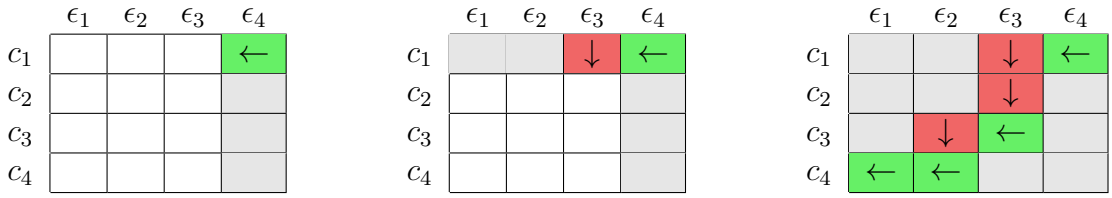

Figure 8.1: Example of different steps of the search method

dominated solutions to the problem. Additionally, the procedure that checks if a feasible solution exists must be modified to account for Coverage.

8.1.1 Search Method

Regarding the search for the non-dominated solutions of the ϵ-indicator and Coverage, we use an approach for search in a 2D space, where each dimension represents an indicator. As in the ϵ-indicator problem, we start by generating the indicator values for pairs of elements, and then merge and sort that matrix into a list of possible values, both for ϵ-indicator and Coverage, using the methods explained in Chapter 4.

The search method starts by considering the highest possible value for ϵ-indicator and the lowest possible Coverage value and finding new solutions by decreasing the value of ϵ-indicator or increasing the Coverage, according to the solution for the indicator values being considered.

For a given pair of values, the algorithm determines if there is a subset such that its ϵ-indicator and Coverage do not exceed the threshold values. Formally, we want to find a subset $R \in B$, $|R| = k$, such that $I_\epsilon(R, B) \leq t_\epsilon$ and $I_C(R, B) \leq t_C$, where t_ϵ and t_C are the threshold values for ϵ-indicator and Coverage, respectively.

Now we have two cases we need to look at. If there is a feasible solution for these thresholds, then increasing the value of Coverage will only produce dominated pairs of values (since the ϵ-indicator value is maintained). Therefore, if there is a feasible solution, the algorithm decreases the value of ϵ-indicator. The second case is if there is no feasible solution for the threshold values. In this case, decreasing the value of ϵ-indicator further restricts the set covering procedure, and does not produce feasible solutions. Therefore, for that value of the Coverage indicator, lower ϵ-indicator produce solutions that are not feasible, so the algorithm increases the Coverage value instead.

In Figure 8.1, these two cases are illustrated. The first solution we find is feasible (green), for $t_\epsilon = \epsilon_4$ and $t_C = c_1$. Therefore, we know that the column will consist of dominated solutions, and consequently move to the left, that is, we decrease the value of ϵ-indicator. For $t_\epsilon = \epsilon_3$ and $t_C = c_1$, our next position, the solution is not feasible (red). Therefore, we know that the solutions we would find to its left would also not be feasible, and consequently move down. Repeating the process, we eventually reach one of the edges of the matrix, and the search ends. At most, we explore n^2 non-feasible solutions and n^2 feasible solutions, since each of these makes the position move to the left or down, and n^2 is the maximum number of rows and columns.

In other words, we start with extreme values for both indicators, and at each step we exclude a value of Coverage or a value of ϵ-indicator from the search. Since there are $\mathcal{O}(n^2)$ values for ϵ-indicator and Coverage, we need to check $\mathcal{O}(n^2)$ pairs of values to

obtain all the solutions. Additionally, given that the algorithm visits the pairs of values in increasing order of Coverage, it is simple to check which solutions are non-dominated.

8.1.2 Set Covering Procedure

The procedure used in the original algorithm checks if there is a feasible solution for a given value of ϵ-indicator. In order to be used on this problem, this procedure must be adapted so that it considers a pair of ϵ-indicator and Coverage values. Moreover, we proved in Propositions 4.2.2 and 6.1.1 that the rows of the adjacency matrices for both ϵ-indicator and Coverage have the "consecutive ones property", as is demonstrated in Proposition 6.1.1.

Therefore, we simply need to adapt the algorithm so that it uses both indicators instead of just one. It is important to remark that even though the chosen elements must cover the entire set considering both ϵ-indicator and Coverage, it is not necessary that each selected element covers the same elements considering both the indicators.

Using the same notation as in Section 4.1, one simple way to solve this problem is to double the reference set on the graph G_i so that each node appears twice, one for the edges of each indicator. This reduces the problem of finding the subset with a pair of values to finding a subset with k elements given an adjacency matrix with size $2n \times n$. Formally, we generate adjacency matrices for the ϵ-indicator and Coverage, join them vertically, and solve the subset selection problem on the new matrix. Since each row of the original matrices has C1P, the rows of the new matrix also have the same properties, so the algorithm may be applied to this matrix.

Another way to obtain the same results, without doubling the graph, is to slightly alter the algorithm so it considers both indicators simultaneously, selecting the next element according to both matrices. This method has the advantage of not needing the operations for reordering the rows present in the original algorithm by Schöbel [11].

This new algorithm starts by considering the first row, and selecting the rightmost column that has a one in that row, on both matrices. This column represents one of the elements of the subset. Then, for each of the two matrices, the rows that have a one in that column are skipped, since the elements corresponding to those rows are already covered. At this time, the algorithm may be operating on different rows on the two matrices. After skipping these rows, the process is repeated to choose the next element, until all rows are covered or it finds that the set cannot be covered.

In the following, we prove that this algorithm is correct, that is, that picking the rightmost column that has a one is the best strategy. First of all, we remark that we need to choose a column that has a one in both columns, since that element must cover the first element considering both ϵ-indicator and Coverage. Therefore, we just need to prove that the rightmost possible column is better than any other, for both ϵ-indicator and Coverage. Since this algorithm still finds the solution in linear time, the complexity for this procedure is still $\mathcal{O}(n \log n)$, considering the overhead caused by the generation of the compressed adjacency matrix.

Proposition 8.1.1. *Let $a, a', b \in B$, with $a'_1 < a_1$ and such that b is covered by both a and a', that is, $\epsilon(a, b) \leq t_\epsilon$, $\epsilon(a', b) \leq t_\epsilon$ and for Coverage, $||a - b|| \leq t_C$, $||a' - b|| \leq t_C$. Then there is no $b' \in B$, with $b'_1 > b_1$, such that b' is covered by a', but not by a, that is, $\epsilon(a', b') \leq t_\epsilon$ and $||a' - b'|| \leq t_C$, but $\epsilon(a, b') > t_\epsilon$ or $||a - b'|| > t_C$.*

Proof. We proceed by contradiction, thereby assuming that such b' exists. Given that $\epsilon(a,b) \leq t_\epsilon$ and $\epsilon(a,b) = \max(b_1/a_1, b_2/a_2)$, we know that $b_2/a_2 \leq t_\epsilon$, and given that $b'_2 < b_2$, we have that $b'_2/a_2 < b_2/a_2$, which implies that $b'_2/a_2 < t_\epsilon$.

Similarly, since $\epsilon(a',b') \leq t_\epsilon$ and $\epsilon(a',b') = \max(b'_1/a'_1, b'_2/a'_2)$, then $b'_1/a'_1 \leq t_\epsilon$. We know that $a'_1 < a_1 \Rightarrow 1/a'_1 > 1/a_1 \Rightarrow b'_1/a'_1 > b'_1/a_1$, which implies that $b'_1/a_1 < t_\epsilon$. Since $b'_1/a_1 < t_\epsilon$, $b'_2/a_2 < t_\epsilon$, we conclude that $\epsilon(a,b') < t_\epsilon$, which is false.

We now look at the second part, that is, $||a - b'|| > t_C$, for any p-norm. Let us first consider the case in which $a_1 > b_1$ (and, consequently, $a_2 < b_2$): Given that $b'_1 > b_1$, then $a_1 - b'_1 < a_1 - b_1$, and since both terms are positive, this implies that $|a_1 - b'_1|^p < |a_1 - b_1|^p$. We also know that $b'_2 < b_2$, so $b'_2 - a_2 < b_2 - a_2$, and consequently $|b'_2 - a_2|^p < |b_2 - a_2|^p$, since both terms are positive. We conclude that $|a_1 - b'_1|^p + |b'_2 - a_2|^p < |a_1 - b_1|^p + |b_2 - a_2|^p < t_C$, and consequently $||a - b'|| < t_C$. For the ∞-norm, $||a - b||_\infty = \max(|a_1 - b_1|, |a_2 - b_2|)$, it is enough that $|a_1 - b'_1| < |a_1 - b_1|$ and $|b'_2 - a_2| < |b_2 - a_2|$, which is a consequence of the case $p = 1$.

In the remaining case, $a_1 \leq b_1$ (and $a_2 \geq b_2$), we apply a similar reasoning. We know that $a'_1 < a_1 \Rightarrow b'_1 - a'_1 > b'_1 - a_1 \Rightarrow |b'_1 - a'_1|^p > |b'_1 - a_1|^p$, since $b'_1 \geq a_1$ and, consequently, the terms are non-negative. For the second coordinate, we use the fact that $a'_2 > a_2$ to conclude that $a'_2 - b'_2 > a_2 - b'_2$, and since the terms are non-negative, we have $|a'_2 - b'_2|^p > |a_2 - b'_2|^p$. We conclude that $|a_1 - b'_1|^p + |b'_2 - a_2|^p < |a'_1 - b'_1|^p + |b'_2 - a'_2|^p < t_C$, and consequently $||a - b'|| < t_C$. Since $|b'_1 - a'_1| > |b'_1 - a_1|$ and $|a'_2 - b'_2| > |a_2 - b'_2|$, as a consequence of $p = 1$, then the same reasoning also holds for the ∞-norm.

In either case, we conclude that the statement is false, and since both $\epsilon(a,b') > t_\epsilon$ and $||a - b'|| > t_C$ are false, we reach a contradiction. □

8.1.3 Time Complexity

As mentioned before in Section 8.1.1, we may need to visit $\mathcal{O}(n^2)$ pairs of values. Using the notation analogous to the one defined in Section 4.1, this algorithm has an increased $T_S(n) = \mathcal{O}(n^2)$, while maintaining $T_P(n) = \mathcal{O}(n^2 \log n)$, which is the complexity of sorting the list of values for both indicators; and $T_C(n) = \mathcal{O}(n \log n)$. Therefore, the final complexity for this algorithm is $\mathcal{O}(n^3 \log n)$.

8.2 Dynamic Programming

Compared to the Threshold Algorithm, the algorithm using Dynamic Programming, only needs to be changed in one particular place: the storage of solutions. Since there is no longer a unique solution for each subproblem, each entry of the matrix must keep a set of non-dominated solutions. Therefore, we use a solution that is similar to that used by Paquete et al. [8].

The remaining problem is how to combine the algorithms for ϵ-indicator and Coverage. However, since the same algorithm can be used for both indicators, we simply need to change it slightly to calculate both indicators, instead of just one. Additionally, instead of choosing the best subproblems, we consider every possible sub-problem, and for each of those, each non-dominated solution.

Formally, the base case is now $T(1,j) = (\epsilon(b_j, b_n), ||b_j - b_n||)$ and we have:

$$T(i,j) = \min_{j<\ell\leq n-i+2} \left(\left\{ \left(\max\left(\delta_{\epsilon,j,\ell}, e\right), \max\left(\delta_{C,j,\ell}, c\right)\right) : (e,c) \in T(i-1,\ell)\right\}\right)$$

where min represents the set of non-dominated vectors. Values $\delta_{\epsilon,j,\ell}$ and $\delta_{C,j,\ell}$ represent the values of $\delta_{j,\ell}$ for each of the indicators, as defined in Sections 5.2 and 6.2.

The final correction needs to be applied to every solution, so we get the final solutions given by:

$$\min_{1\leq j\leq n-k+1} \left(\left\{ \left(\max\left(\epsilon(b_1, b_j), e\right), \max\left(||b_1 - b_j||, c\right)\right) : (e,c) \in T(k,j)\right\}\right)$$

From these recursion equations, it is clear that we cannot simply apply the improvement described in Section 5.2.1. Given that we do not want to find an optimal solution but a set of non-dominated ones, we cannot simply look at one value of ℓ. Choosing just one value of ℓ may yield the optimal solution for one of the indicators, but it may not give all of the compromise solutions we want the algorithm to find. Therefore, we do not use this improvement in any multiobjective algorithm.

We remark that we need a data structure to store the set of non-dominated vectors efficiently. We chose to use an AVL binary balanced tree (see Kung et al. [7]). Since we need to be able to efficiently iterate over the ordered list of vectors (according to the first coordinate), a binary tree is a better solution than a hash table.

Now, we need to define an operation to add a 2D vector to the non-dominated set. This operation first needs to check if the vector is not dominated, and then remove vectors in the set that are dominated by the new vector. Checking if the vector is dominated is quite simple: we just need to check if the first vector to its left dominates it. The newly dominated vectors are also easy to find, since the vectors to the left of the inserted vector have a smaller first coordinate, and as such cannot be dominated by it. Therefore, we simply need to check the vectors immediately to its right.

Formally, the vectors in the tree are denoted as $p_1, p_2, p_3, \ldots, p_m$, sorted by the first coordinate and we want to insert vector q, then we find

$$i = \max\left(\left\{j : p_j^{(1)} < q^{(1)} \vee \left(p_j^{(1)} = q^{(1)} \wedge p_j^{(2)} < q^{(2)}\right)\right\}\right)$$

If there is no such i or if p_i does not dominate q, then q is non-dominated. Now we need to find the largest j such that $j > i$ and $p_j^{(2)} > q^{(2)}$, considering $i = 0$ if it was not found in the previous. If such a j exists, the dominated vectors we need to remove from the set are $\{p_{i+1}, \ldots, p_j\}$.

Regarding the time complexity of this algorithm, we have to consider that there is a matrix of size $\mathcal{O}(k\,n)$ that needs to be filled. For each of the elements of the matrix, $T(i,j)$, we check $\mathcal{O}(n)$ values of ℓ in the range $j < \ell \leq n - i + 2$, and for each of the vectors, stored in $T(i-1,\ell)$, we must add it to our set. Since each set may have at most $\mathcal{O}(n^2)$ vectors, since these are the possible values for each of the indicators, the number of operations of insertion and lookup, to iterate over the vectors on one set and insert them in the other is, for each value of ℓ, $\mathcal{O}(n^2 \log n)$. Therefore, the final complexity is $\mathcal{O}(k\,n^4 \log n)$. We remark that this is a worst case scenario, and that the algorithm may have a lower average time complexity.

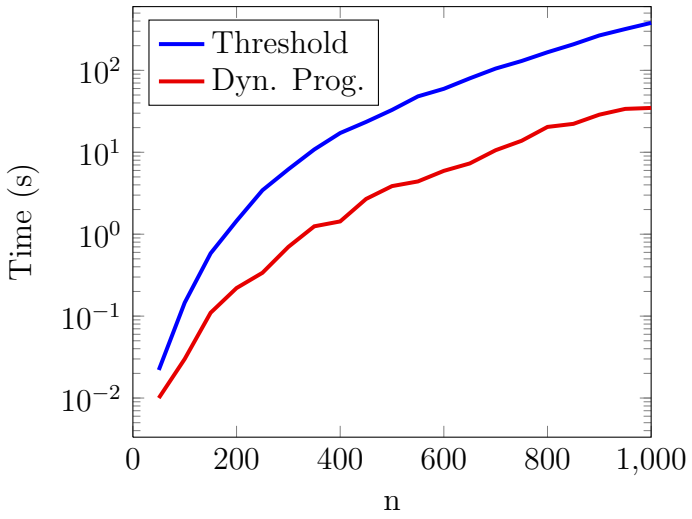

Figure 8.2: Running time for the two algorithms with $50 \leq n \leq 1000$, $k = 20$

8.3 Complexity and Results

In this chapter we present two different algorithms that solve the problem discussed, whose complexities are:

- Threshold Algorithm $\mathcal{O}(n^3 \log n)$
- Dynamic Programming $\mathcal{O}(k\, n^4 \log n)$

8.3.1 Experimental Results

Two tests were run, in order to compare the performance of the two algorithms and the influence of the parameters n and k on the running time. The setup of these tests is similar to that described in previous chapters, and is discussed in more detail in Chapter 3.

The results for the first test are presented in Figure 8.2. These results clearly indicate that the Dynamic Programming algorithm performs better than the Threshold Algorithm, taking approximately 10 times less to solve the instance.

These results are not consistent with the complexities of the presented algorithms. However, as we said in Section 8.2, the presented complexity is an upper bound considering the maximum number of non-dominated vectors that can be present in a entry of the matrix. Therefore, it is possible that the Dynamic Programming approach has better performance, if there are few non-dominated vectors per entry.

Another important factor is the value of k, since the time complexity of the Dynamic Programming version depends linearly on this factor. The comparison for the second test, varying the value of k, is presented in Figure 8.3. First of all, the Threshold Algorithm takes approximately constant time when varying k, whereas the Dynamic Programming version seems to increase with k. This is consistent with the complexities of the algorithms, and was expected. This also means that if we consider a large enough value for k, the Threshold Algorithm takes less time.

This is not visible in the graph because the Dynamic Programming version has no registered times beyond $k = 300$, indicating that the algorithm failed to complete. The

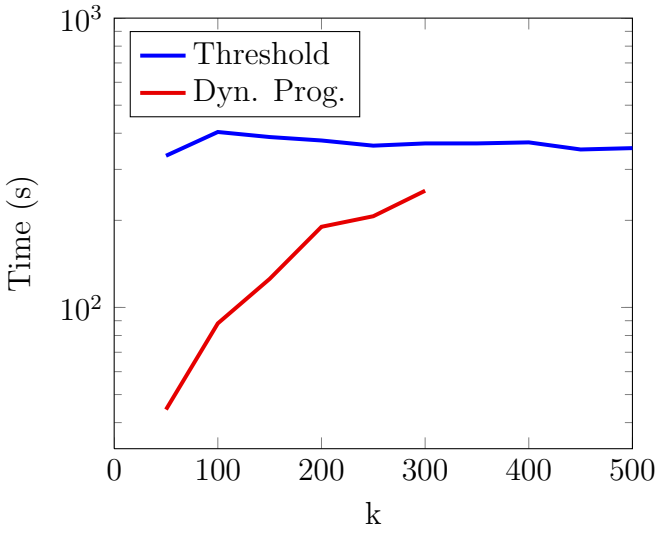

Figure 8.3: Running time for the two algorithms with $n = 1000$, $50 \leq k \leq 500$

reason for this is that, even though memory is freed as soon at is no longer needed, the implementation uses too much memory, and quickly exhausts the 4GB of RAM that are available, crashing the executions. Naturally, this makes the Threshold Algorithm more adequate for high values of k or in cases where a limited amount of memory is available, since it has $\mathcal{O}(n^2)$ space complexity.

Chapter 9

Representation Problem using ϵ-indicator and Uniformity

In this chapter, we define a new problem, based on Uniformity and ϵ-indicator. As we discussed in Chapter 7, this indicator measures the distance between the chosen elements, unlike the ϵ-indicator, which compares the chosen elements with the whole set.

The problem we discuss consists in finding the subset with a given cardinality that minimises the ϵ-indicator and maximises Uniformity. Uniformity is useful as an indicator because it tries to maximise the distance between elements, therefore yielding subsets that are "spread out" over the entire set. Combining this property with the ϵ-indicator, results in the algorithm discovering representative subsets, that are close to the original set, but whose elements are diversified.

Formally, given a set of non-dominated vectors B and an integer k, the objective of this problem is to find subsets $R \subseteq B$, with $|R| = k$ that maximise the value of Uniformity and minimise the value of the ϵ-indicator. In order to achieve some consistency, we switch the sign of the Uniformity value and minimise this. Therefore, we have:

$$\underset{\substack{R \subseteq B \\ |R|=k}}{\arg\min} \; (I_\epsilon(R, B), -I_U(R))$$

As in Chapter 8, the arg min operator returns the non-dominated solutions.

In this chapter, we present two alternatives to solve this problem, based on the algorithms previously discussed in Chapter 5. Similarly to Chapter 8, we detail the changes needed to adapt the algorithms to use the two indicators together.

9.1 Threshold Algorithm

Similarly to the algorithm described in Section 8.1 using the ϵ-indicator and Coverage, two main parts of the Threshold Algorithm need to be adapted: the search method; and the set covering procedure, which finds a subset for a given pair of threshold values.

First of all, the search method needs to be adapted for two objectives. Since knowledge about the indicators is not necessary, we can, in fact, reuse the same strategy used for the ϵ-indicator and Coverage problem. Therefore, we refer to Section 8.1.1 for a detailed discussion of this search method.

To account for the fact that we want to maximise Uniformity, we start on the lowest possible value of both ϵ-indicator and Uniformity. Additionally, if we find a feasible solution, we increase the value of Uniformity, and if no feasible solution is found, we decrease the value of ϵ-indicator.

9.1.1 Set Covering Procedure

Unlike Coverage and ϵ-indicator, which share a similar structure, Uniformity is different than these operators, since it measures the distances between chosen elements. Therefore, we must use an alternative algorithm to solve the individual problems, and then combine these algorithms to solve the multiobjective version.

The approaches described in Section 7.1, for Uniformity, and 5.1.1, for ϵ-indicator, both use the equivalence between paths in a graph and the chosen subsets. We now show an algorithm that combines these two approaches to solve the multiobjective problem.

Since each subset is represented as a path in each of the graphs, then in order to solve both problems we must get a path that is valid in both graphs. Therefore, we can simply intersect the set of arcs, which results in a new graph $G_{\epsilon,U}$. By finding a path in this graph, we find a path in both G_U, and G_ϵ, therefore finding a subset R such that $I_\epsilon(R, B) \leq t_\epsilon$ and $I_U(R) \leq t_U$.

However, Uniformity and ϵ-indicator are different, in the sense that adding elements to a subset may decrease the Uniformity, while removing elements may increase the ϵ-indicator. Therefore, we must find a path with exactly $k + 2$ nodes, including h_s and h_t. To solve this problem, we build the graph as described, and then visit each node, in the order $h_s, h_1, \ldots, h_n, h_t$, saving the possible lengths of the paths from h_s to that node. When a node is visited, all the outgoing arcs from that node are also visited, and the destination nodes are updated. Fortunately, we only need to keep the size of the shortest path and longest path, because there is a path for every size between those two values (see Proposition A.1.1 in appendix).

After visiting every node, the algorithm is able to check if there is a path with $k + 2$ nodes, as it is enough to check if $k + 2$ is between the values stored in h_t. As this corresponds to a subset with cardinality k, the algorithm can then return if an acceptable subset exists for the given threshold values, as is required for the algorithm to work correctly.

9.1.2 Time Complexity

This new Threshold Algorithm, for ϵ-indicator and Uniformity, uses the same search method we described in Section 8.1.1. Therefore, we also have $T_S(n) = \mathcal{O}(n^2)$, for this algorithm.

As to the set covering procedure we described, the algorithm has four essential tasks:

- Calculating the adjacency matrix for the ϵ-indicator, which takes $\mathcal{O}(n \log n)$;
- Calculating the s_j and e_j for the matrix, with the positions of the first and last values of 1, in $\mathcal{O}(n^2)$;
- Generating the graph, which can be done in $\mathcal{O}(n^2)$ by checking every pair of elements;

- Finding the lengths of the paths in the graph, which can be done in $\mathcal{O}(n^2)$ since we visit each node and arc once.

In conclusion, this set covering procedure has a time complexity of $T_C(n) = \mathcal{O}(n^2)$, Therefore, the final complexity for this algorithm is $\mathcal{O}(n^4)$.

9.2 Dynamic Programming

If we consider the Dynamic Programming algorithm presented in Section 8.2, there are only some details that need to be modified, which will allow the algorithm to use Uniformity instead of Coverage. The same structure, with a set of non-dominated solutions in each entry of the matrix, as well as the use of AVL trees, are kept in this algorithm.

In order to reuse the reasoning and implementation of the algorithm with ϵ-indicator and Coverage, this version will minimise $-I_U(R)$, which is equivalent to maximising $I_U(R)$.

The Dynamic Programming algorithm is based on adding an element to a subset of size $i-1$ to obtain a subset of size i. As we discussed in Section 7.2, this idea can be applied to the Uniformity indicator.

Therefore, we can use the same approach to solve this problem, by considering $T(1,j) = (\epsilon(b_j, b_n), -\infty)$ and by calculating $T(i,j)$ using the following expression:

$$T(i,j) = \min_{j<\ell\leq n-i+2} \left(\{(\max\left(\delta_{\epsilon,j,\ell}, e\right), -\min\left(||b_j - b_\ell||, -u\right)) : (e,u) \in T(i-1,\ell)\}\right)$$

where min represents the set of non-dominated vectors and $\delta_{\epsilon,j,\ell}$ represents the value of $\delta_{j,l}$ as defined in Section 5.2.

Similarly, since the Uniformity does not need a final correction, the solutions for the problem are given by:

$$\min_{1\leq j\leq n-k+1} \left(\{(\max\left(\epsilon(b_1, b_j), e\right), u) : (e,u) \in T(k,j)\}\right)$$

The time complexity for this algorithm is similar to the algorithm described in Section 8.2, since the algorithm executes similar steps to obtain the solutions. The same array of size $\mathcal{O}(k\,n)$ needs to be filled, and there are $\mathcal{O}(n)$ subproblems to check, represented by the range $j < \ell \leq n - i + 2$. Also similarly, each entry may contain $\mathcal{O}(n^2)$ non-dominated vectors, which leads to a complexity of $\mathcal{O}(n^2 \log n)$ to iterate over them and add them to the new set. This results in a final complexity of $\mathcal{O}(k\,n^4\,log\,n)$.

9.3 Complexity and Results

In this chapter we presented two different algorithms that solve the problem discussed, whose complexities are:

- Threshold Algorithm $\mathcal{O}(n^4)$
- Dynamic Programming $\mathcal{O}(k\,n^4 \log n)$

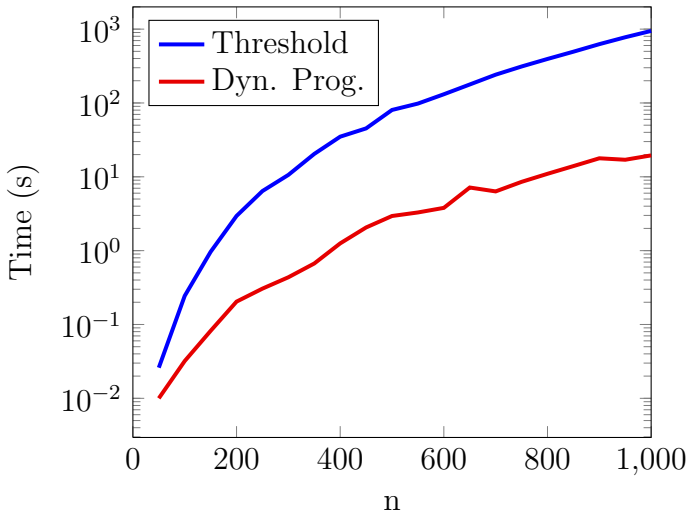

Figure 9.1: Running time for the two algorithms with $50 \leq n \leq 1000$, $k = 20$

9.3.1 Experimental Results

In order to compare the performance of the two algorithms, we run two tests, testing the influence of the parameters n and k on the running time. The experimental setup is similar to that of previous chapters, and is presented in more detail in Chapter 3.

In Figure 9.1, the results for the first test are presented. Similarly to the results discussed on Section 9.3.1, the results indicate that the Dynamic Programming version performs much better than the Threshold Algorithm.

These results are not consistent with the complexities, since we would expect that the algorithm with lower complexity would perform better. However, the difference in complexities is simply $k \log n$, and in this test k is constant. Moreover, the complexity for the Dynamic Programming version represents an upper bound for the worst case scenario, considering the maximum number of non-dominated vectors stored in each entry. If the size of the set is lower than the maximum value, then that will reflect on the running time, explaining the difference in performances of the algorithms.

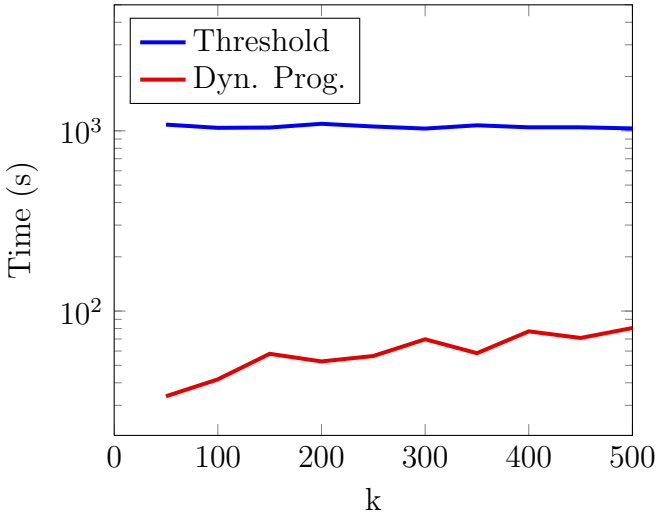

Figure 9.2: Running time for the two algorithms with $n = 1000$, $50 \leq k \leq 500$

The second test, on the other hand, does not consider k as a constant value, and may present different results. The performances for the two algorithms, with a fixed value of n and varying k, are present in Figure 9.2.

However, even when considering the influence of k, and even though the Dynamic Programming increases with k, it still has a much greater performance than the Threshold Algorithm. Also, even though the Dynamic Programming version uses more memory, due to the fact that it needs to store the non-dominated solutions for every subproblem, that does not affect its performance. In conclusion, unlike in the results in Section 9.3.1, this version of the Dynamic Programming algorithm does not crash for large values of k, and has better performance than the Threshold Algorithm, independently of the value of k.

Chapter 10

Triobjective Representation Problem

In this chapter, we introduce a new problem, which results from joining the problems of Chapters 8 and 9. This is the triobjective version of the representation problem, with the goal of minimising the ϵ-indicator and Coverage and maximising the Uniformity.

Formally, given a set of non-dominated vectors B and an integer k, the objective of this problem is to find:

$$\underset{\substack{R \subseteq B \\ |R|=k}}{\arg\min} \; (I_\epsilon(R,B), I_C(R,B), -I_U(R))$$

As in previous chapters, the arg min operator takes a multiobjective meaning and consequently returns the non-dominated solutions.

We present two alternatives, adapted from the biobjective algorithms presented in Chapters 8 and 9, based on the Threshold Algorithm and Dynamic Programming. We present the modifications necessary to use those algorithms to solve this problem, and then discuss the performance of the two algorithms, with the support of experimental results.

10.1 Threshold Algorithm

As with the other multiobjective algorithms, described in Chapters 8 and 9, to use the Threshold Algorithm for the three indicators we need to adapt the search method, as well as the set covering procedure.

For the search method, we chose to adapt the two dimensional version, by fixing one of the indicators. In other words, for each value of the ϵ-indicator, the algorithm executes the search over the values of Coverage and Uniformity, similarly to the method described in Section 9.1.

For the set covering procedure, we adapted the version described in Section 9.1.1. This time, we add Coverage, and therefore add a new graph, G_C, which is analogous to G_ϵ, as described in Section 6.1. The graph used for the algorithm, $G_{\epsilon,C,U}$ is defined as the graph whose edges are the intersection of the edges of G_ϵ, G_C, G_U.

The proof of the correctness of the algorithm for ϵ-indicator and Uniformity is also valid when adding Coverage, since the proof does not use any specific knowledge about the ϵ-indicator other than the adjacency matrix having the "consecutive ones property".

However, we have already seen that the Coverage indicator has many of the properties of the ϵ-indicator, including C1P for the rows and columns of the adjacency matrix, as stated in Propositions 6.1.1 and 6.1.2, respectively.

Compared to the ϵ-indicator and Uniformity version, this algorithm has some overhead because of the extra operations to join G_C to the other graphs. Nevertheless, the complexity for this procedure is the same, $\mathcal{O}(n^2)$.

Finally, we discuss the method to store and filter the non-dominated solutions. Despite being relatively simple when the search space is two dimensional, when we add a third dimension, we cannot simply save the solutions in order, allowing us to quickly check if a solution is dominated.

A solution is described by Kung et al. [7] to extract non-dominated solutions from a set, concretely for three dimensions. This solution requires that the solutions are inserted in lexicographical order in a binary tree of two dimensional vectors. If a 3D vector, with its first coordinate removed, is dominated by any vector of the tree, then we know that the 3D vector is also dominated. If it is not dominated by any vector of the tree, then we insert it in the tree and we know that the 3D vector is also non-dominated. This algorithm has a $\mathcal{O}(m \log m)$ complexity, where m is the number of vectors.

Our algorithm naturally iterates over the different indicators in an ordered fashion, so this algorithm allows us to keep the solutions without changing its structure too much. Our implementation uses an AVL tree to store the 2D vectors and an ordinary array to store the 3D vectors. Moreover, the implementation tests each solution as soon as it finds it, so as to not waste memory with dominated solutions.

Given that we have, at most, n^4 solutions (one for each pair of ϵ-indicator and Uniformity), then we must add $\mathcal{O}(n^4 \log n^4) = \mathcal{O}(n^4 \log n)$ to the complexity. Given that the search method has complexity $T_S(n) = \mathcal{O}(n^4)$ and the set covering procedure $T_P(n) = \mathcal{O}(n^2)$, even after adding the extra complexity of $\mathcal{O}(n^4 \log n)$ for keeping only the non-dominated solutions, we have a final complexity of $\mathcal{O}(n^6)$.

10.2 Dynamic Programming

The Dynamic Programming algorithm for this problem is based on the biobjective versions described in Sections 8.2 and 9.2. In order to work with three objectives, we must, first of all, modify the algorithm so that is calculates and stores the vectors with the three objectives. Therefore, we now have $T(1,j) = (\epsilon(b_j, b_n), ||b_j - b_n||, -\infty)$ and:

$$T(i,j) = \min_{j<\ell\leq n-i+2} \left(\left\{ \left(\max\left(\delta_{\epsilon,j,\ell}, e\right), \max\left(\delta_{C,j,\ell}, c\right), -\min\left(||b_j - b_\ell||, -u\right)\right) : (e,c,u) \in T(i-1,\ell)\right\}\right)$$

with $\delta_{\epsilon,j,\ell}$ and $\delta_{C,j,\ell}$ as defined in Section 8.2.

Additionally, the final solutions are generated with:

$$\min_{1\leq j\leq n-k+1} \left(\left\{ \left(\max\left(\epsilon(b_1, b_j), e\right), \max\left(||b_1 - b_j||, c\right), u\right) : (e,c,u) \in T(k,j)\right\}\right)$$

Using these expressions, the algorithm is able to find all the non-dominated solutions. However, we can no longer use an AVL as we did for the previous chapters, since we have three dimensional vectors. Kung et al. [7] suggest a solution for extracting the

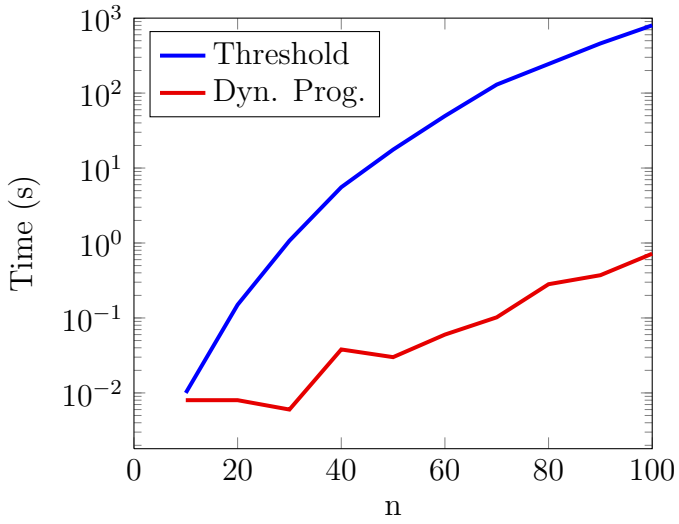

Figure 10.1: Running time for the two algorithms with $10 \leq n \leq 100$, $k = 20$

non-dominated solutions, but the list of vectors must be sorted according to the first coordinate. Therefore, the algorithm sorts the values according to the first coordinate, and then uses the procedure to extract the non-dominated vectors, similarly to what is done in Section 10.1.

In practice, this method of removing the dominated vectors is less efficient, since a vector of solutions must be kept, and only then are the dominated solutions removed. Nevertheless, the complexity is $\mathcal{O}(m \log m)$, where m is the number of solutions. Since we may have $\mathcal{O}(n^4)$ solutions for each subproblem, sorting and removing the dominated solutions has $\mathcal{O}(n^4 \log n)$ time complexity. This is repeated for $k\,n$ entries of the matrix, and for each of them, at most $\mathcal{O}(n)$ problems need to be checked. Therefore, the time complexity of the algorithm is $\mathcal{O}(k\,n^6 \log n)$. As we mentioned in previous chapters, this is the complexity for the worst case, and therefore it may not reflect on the running time.

10.3 Complexity and Results

In this chapter we presented two different algorithms that solve the problem discussed, whose complexities are:

- Threshold Algorithm $\mathcal{O}(n^6)$
- Dynamic Programming $\mathcal{O}(k\,n^6 \log n)$

10.3.1 Experimental Results

In order to compare the performance of the two algorithms, we run two tests, testing the influence of the parameters n and k on the running time. The experimental setup is similar to that of previous chapters, and is presented in more detail in Chapter 3.

The complexities of these algorithms are at least $\mathcal{O}(n^6)$, so a small value such as $n = 100$ is already a challenge for current computers. In Figure 10.1, the results for the first test are presented. Since the algorithms have very different running times, we chose to represent the data using a logarithmic scale. It is clear that the Dynamic Programming

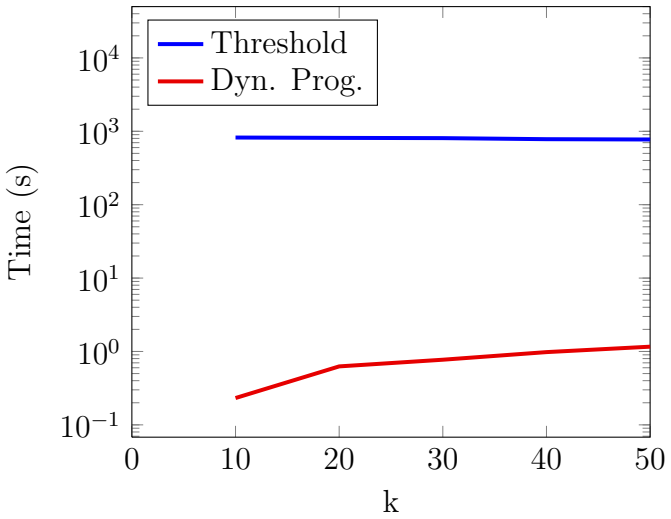

Figure 10.2: Running time for the two algorithms with $n = 100$, $10 \leq k \leq 50$

approach performs much better, reaching a difference of 10^3 around $n = 70$.

As we mentioned in previous chapters, the complexity of the Dynamic Programming version is an estimate for the worst case scenario, whereas the complexity for the Threshold Algorithm is much tighter. This explains why one of the algorithms takes more than 100 seconds to solve an instance, while the other does it in less than 1 second.

The second test, with the goal of studying the influence of k on the running time, confirms the superiority of the Dynamic Programming version. Although the Dynamic Programming version shows an increase in running time for larger values of k, it still outperforms the Threshold Algorithm by a large margin. This increase is consistent with the complexity of the Dynamic Programming algorithm, as is the absence of influence to the Threshold Algorithm.

Even though the Dynamic Programming version performs much better, it is not the perfect solution. As we mentioned before, this approach stores a great number of solu-

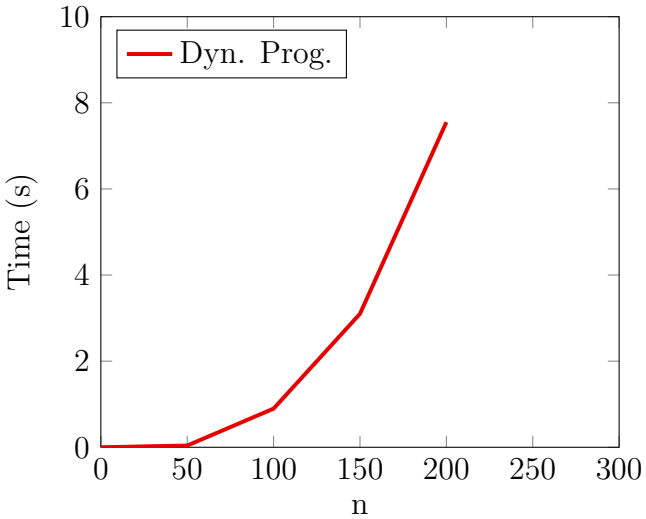

Figure 10.3: Running time for the Dynamic Programming approach with $10 \leq n \leq 300$, $k = 20$

tions, whereas the Threshold Algorithm does not.

In Figure 10.3, we present a third test, in which we measured the running time for greater values of n for the Dynamic Programming algorithm. Similarly to what we described in Section 8.3.1, this algorithm stops working in certain instances, since it generates too many solutions and takes up all the available memory. Even though the algorithm solved some instances in reasonable time, any instance with $n > 200$ causes a crash in the program.

In conclusion, the Dynamic Programming approach clearly performs better than the Threshold Algorithm. However, whereas the Threshold Algorithm only uses memory to store the values of the indicators to test, and the necessary space to keep the final solutions, the Dynamic Programming uses much more memory, which makes it impractical to use in a modest configuration, for large values of n.

Chapter 11

Conclusion

In this thesis, we started by presenting an algorithm to solve the subset selection problem, by Ponte et al. [9], and then presented our contributions: three improvements to the original algorithm, as well as a proof of correctness. The experimental results we presented also showed that one of our improved versions performed consistently better than the original algorithm.

We then introduced the representation problem, which is a more specific version of the subset selection problem, and proposed two different algorithms to solve that problem, with regard to the ϵ-indicator. We denoted these algorithms by Threshold Algorithm, which was adapted from the best approach for the subset selection problem; and Dynamic Programming algorithm. In the following chapters, we introduced two more indicators, Coverage and Uniformity, and described the new versions of the two algorithms for these indicators.

The three last chapters also concerned the representation problem, but in its multi-objective version. In these problems, we want to optimise a combination of indicators, instead of just one. For these problems, which considered the ϵ-indicator with Uniformity and/or Coverage, we also presented our algorithms, based on the single objective versions.

All of these algorithms were presented with a combination of theoretical and practical results, which ensure that we know that the algorithms are correct, and their asymptotic behaviours, as well as which algorithm performs better in practice.

Throughout this thesis, we found that the Dynamic Programming algorithm performed better, in practice, for the different versions of the representation problem, even when compared to the better version for the subset selection problem. However, particularly on the multiobjective versions, it sometimes uses too much memory and the running time depends on the size of the subset to select, both flaws that the Threshold Algorithm does not have.

11.1 Future Work

Even though we were able to solve all the discussed problems in polynomial time, the proposed algorithms have complexities with relatively large degrees. It may be possible to improve the performance and complexity of the algorithms.

Specifically, the experimental results also suggest that the complexity for the Dynamic

Programming approaches can be theoretically improved, providing a tighter bound and a better understanding of the performance of the algorithm. Another problem with this approach is that is uses too much memory, for large instances. It may be possible to reduce the memory footprint, and therefore make the algorithm more useful. Additionally, the relation between the chosen indicators and the number of non-dominated solutions is not clear. We remark that the non-dominated solutions influence the running time in some of the approaches, such as those using dynamic programming.

As for the Threshold Algorithm, specifically the version for three indicators, it may be possible to reduce the complexity of the search procedure by using the three dimensional equivalent of a binary search. By dividing the search space into octants correctly, it may be possible to exclude two of them immediately, which would leave six octants to be searched in.

One final issue with the presented algorithms is that they are not extendable to three or more dimensions. We know that the general Uniformity value is NP-Hard for three or more dimensions, as per Wang and Kuo [12]. The p-center problem, which is related to the representation problem using Coverage is also NP-Hard for more than two dimensions, as per Garey and Johnson [6]. We conjecture that the representation problem for ϵ-indicator or Coverage is also NP-Hard in general. Even though the approaches we presented are not applicable for sets of points with three dimensions, it may be possible to design algorithms that do it.

Appendix A

Proof of Correctness for ϵ-indicator and Uniformity

In this appendix, we demonstrate that the algorithm presented in Section 9.1 is correct. The most important step to do this is the proof that there are paths of any size between the minimum and the maximum. This is presented in Proposition A.1.1.

A.1 Threshold Algorithm

Proposition A.1.1. *Given a graph $G_{\epsilon,U}$, and two paths, P and Q, represented as sequences of nodes, then for each i, $|Q| < i < |P|$, there is a path P', such that $|P'| = i$.*

Proof. If $|P| \leq |Q|+1$, there are no values of i that satisfy the condition, and the theorem is trivially true.

Let us operate by induction on the difference $d = |P| - |Q|$. Our base case is $d = 1$, a case described above, and our induction hypothesis is that the theorem is true for $|P'| - |Q| < d$.

We then prove, for P and Q such that $|P| - |Q| = d$, that there is a path P' such that $|P'| - |Q| = d - 1$, or equivalently $|P'| = |P| - 1$. Therefore, by applying the induction hypothesis over P' and Q, we know that there is a path for each size i, with $|Q| < i < |P| - 1$. Adding the fact that there is also a path with size $|P'| = |P| - 1$ and we get that there is a path for each size i, with $|Q| < i < |P|$, proving the hypothesis.

This proof will focus on the case where the paths have only the first and last node in common. This is, in fact, sufficient for the general case, since we can partition the paths using the common nodes between two paths. Since the result applies for each of these parts, then it will also apply to the concatenation of these parts or, in other words, to the sums of the lengths of the parts.

For this proof, we need to look at how the nodes of the two paths are ordered, and therefore we considered the sorted sequence of nodes, according to this order: h_s, h_1, ..., h_n, h_t. Since the first and last nodes are common, they only need to appear once in this sorted sequence of nodes.

We now look at a special case: if there are three or more nodes of the path P in a row (in the sorted list), then we can create a path with size $|P| - 1$ by simply removing

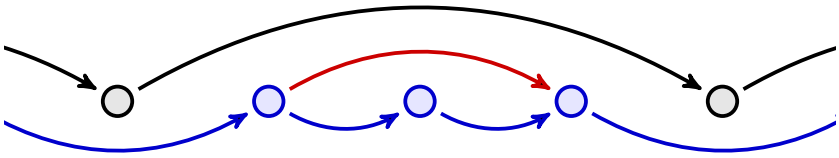

Figure A.1: Representation of the case with three consecutive elements in P

the second node. Let us denote the first three nodes by p_i, p_{i+1}, p_{i+2}, and the let $q_j \in Q$ be the last node before p_{i+1}, with $q_{j+1} \in Q$ being the first note after p_{i+1}. Formally, we create a new path $P' = P \setminus p_{i+1}$, with size $|P| - 1$, and argue that this path is valid.

In Figure A.1, an example of this special case is presented. By removing the second blue node, we remove the two blue arcs connected to it, and insert the red arc to complete the path. If this red arc is valid, the new path is also valid. Moreover, since we removed a node from the path, it has one node less than the original path, as desired.

As we saw before, removing elements from a subset does not degrade Uniformity, and consequently we may remove nodes from the path and keep it valid, according to G_U. Regarding the ϵ-indicator, for each element covered by the element of B corresponding to p_{i+1}, we also know that it must be covered by the elements corresponding to q_j or q_{j+1}. This means that for each such row, there is a value of 1 on columns corresponding to q_j and to p_{i+1}, in which case we know there must be a value of 1 on the column relative to p_i, because the row has the "consecutive ones property". The same reasoning can be applied to for q_{j+1} and p_{i+2}. In conclusion, either the element corresponding to p_i or p_{i+2} must cover the elements previously covered by p_{i+1} and therefore the new path and subset are acceptable.

We now exclude the case above, and therefore consider that there are at most two nodes of P without a node of Q in between. In order to describe the new path for the remaining case, we will adopt a new notation. A new list of numbers will be derived by inserting a 1 for every node that belongs to path P (including common nodes) and -1 for nodes that belong exclusively to Q, in the same order as the sorted list. As an example, $p_1, p_2, q_1, q_2, p_3, q_3, p_4$ becomes $1, 1, -1, -1, 1, -1, 1$. We can further improve this representation by summing the adjacent numbers with the same sign. The example would thus become $2, -2, 1, -1, 1$.

This list, which we denote as ℓ, has some important properties:

- There is no positive number greater that 2, since there are no more than two nodes of P in a row;
- Its sum is at least 4, since the total sum is equivalent to $P - Q + 2$, with the 2 accounting for the common nodes at the beginning and end of the paths;
- If we consider the pairs of consecutive negative and positive numbers, the partial sums only increase if this pair is $(-1, 2)$

To generate the new path, we start by finding the first position such that $L_i = 2$. If a value of 2 does not exist, then all the positive values are 1. Since every positive number (except the last) is followed by a negative number, then the sum would be at most 1, contradicting what we said about the sum being at least 4.

We denote the partial sum up to that point as s, that is, $s = \sum_{j=0}^{i} L_j$. Every value

Figure A.2: Representation of the case with three consecutive elements in P

of 1 before L_i is followed by a negative number, and consequently, $s \leq 2$. Since the total sum is at least 4, the partial sums must increase to the total sum and since the partial sums only increase by 1 for each pair $(-1, 2)$ of consecutive values, there must be an $L_{i'} = 2$, such that $\sum_{j=0}^{i'} L_j = s + 1$.

Given these positions i and i', we define the new path as follows: we start by including the nodes of path P, until we reach the position corresponding to L_i. There, we add the first node from the pair that originated L_i, and then add the next node from Q. Until we reach the position corresponding to $L_{i'}$, we keep adding the nodes from Q. When we reach the position corresponding to $L_{i'}$, we skip the first node from the pair that originated that value, and then add every node from the path P from there on.

An example of this new path is presented in Figure A.2. In this case, to generate the path with one less node, we would start by following the arcs and nodes of the blue path (longer), until we reached two nodes of this path in a row. From the first of these nodes, we would follow the red arc to the next black node, and follow the black arcs. When two blue nodes in a row are reached, if they satisfy the condition mentioned above, we then follow the red arc from the black node to the second blue node, and follow the blue path until the end. By following this method, we argue that the new path is valid, or, in other words, that the red arcs exist in the graph. We also show that the path obtained by this method has one node less than the blue path, that is, has $|P| - 1$ nodes, as desired.

We recall that $\sum_{j=0}^{i'} L_j = s+1$ and $\sum_{j=0}^{i} L_j = s$, implying that $\sum_{j=i+1}^{i'} L_j = 1$. Since $L_{i'} = 2$, we also know that $\sum_{j=i+1}^{i'-1} L_j = -1$. Consequently, if we replace every node in the path P for every node in Q, between the positions corresponding to L_i and $L_{i'}$ (excluding these), we get $|P| + 1$ nodes. However, accounting for the two nodes we skip on the positions relative to L_i and $L_{i'}$, we get the desired $|P| - 1$ nodes.

Formally, we have a path P' of the form $(p_1, \ldots, p_j, q_l, q_{\ell+1}, \ldots, q_{\ell'}, p_{j'}, \ldots, p_{|P|})$. We will now demonstrate that this path is valid, by showing that the arcs (p_j, q_l) and $(q_{\ell'}, p_{j'})$ exist.

Regarding Uniformity: since there is a node p_{j+1} between p_j and q_l on the sorted sequence, and this sequence is sorted by the first coordinate, then by Proposition 6.2.1, the distance between the elements corresponding to elements p_j and p_{j+1} is smaller than between p_j and q_l. Therefore, since (p_j, p_{j+1}) is an arc in G_U, (p_j, q_l) must be too, since it is also above the threshold. Similarly for $q_{\ell'}$, $p_{j'-1}$ and $p_{j'}$, by Proposition 6.2.2, $(q_{\ell'}, p_{j'})$ is an arc in G_U.

For the ϵ-indicator, we only need to concern with the elements covered by the element corresponding to p_{j+1}, for the subset corresponding to P. These elements must be covered either by q_l or $q_{\ell-1}$, since the arc $(q_{\ell-1}, q_l)$ exists in $G_{\epsilon,U}$. Therefore, for any row corresponding to an element covered by p_{j+1}, we either have a value of 1 in the column

for q_l or $q_{\ell-1}$. If there is a value of 1 in the column for q_l, we are done, since the respective element is in the new path. Otherwise, the column p_j must also have a value of 1, since it is between the columns for $q_{\ell-1}$ and p_{j+1} and the row has the "consecutive ones property". The same reasoning may be applied to $q_{\ell'}$, $p_{j'}$ and $p_{j'-1}$, yielding a similar conclusion. Therefore, the new path we described corresponds to a subset which is acceptable, and therefore the path is also valid for G_ϵ.

In conclusion, this new path with $|P| - 1$ nodes is a valid path in both G_ϵ and G_U, and consequently valid in $G_{\epsilon,U}$. Therefore, we proved that there always is a path with $|P| - 1$ nodes, proving the induction hypothesis. □

Bibliography

[1] J. Bader. *Hypervolume-Based Search for Multiobjective Optimization: Theory and Methods*. PhD thesis, ETH Zurich, Switzerland, 2010.

[2] N. Beume, B. Naujoks, and M. Emmerich. SMS-EMOA: Multiobjective selection based on dominated hypervolume. *European Journal of Operational Research*, 181(3):1653–1669, 2007.

[3] B. Chazelle and L. J. Guibas. Fractional cascading: I. A data structuring technique. *Algorithmica*, 1(2):133–162, 1986.

[4] K. Deb and A. Pratap. A fast and elitist multiobjective genetic algorithm: NSGA-II. *IEEE Transactions on Evolutionary Computation*, 6(2):182–197, 2002.

[5] C. M. Fonseca and P. J. Fleming. Genetic algorithms for multiobjective optimization: formulation, discussion and generalization. In *Proceedings of the fifth International Conference on Genetic Algorithms*, pages 416–423, 1993.

[6] M. Garey and D. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. A Series of Books in the Mathematical Sciences. W. H. Freeman, 1979.

[7] H. T. Kung, F. Luccio, and F. P. Preparata. On finding the maxima of a set of vectors. *Journal of the Association for Computing Machinery*, 222(4):469–475, 1975.

[8] L. Paquete, C. M. Fonseca, K. Klamroth, and M. Stiglmayr. Concise representation of nondominated sets in discrete multicriteria optimization. In *Proceedings of the 21st International Symposium on Mathematical Programming*, ISMP 2012, page 95, 2012.

[9] A. Ponte, L. Paquete, and J. R. Figueira. On beam search for multicriteria combinatorial optimization problems. In N. Beldiceanu, N. Jussien, and E. Pinson, editors, *CPAIOR*, volume 7298 of *Lecture Notes in Computer Science*, pages 307–321. Springer, 2012.

[10] S. Sayin. Measuring the quality of discrete representations of efficient sets in multiple objective mathematical programming. *Mathematical Programming*, 87(3):543–560, 2000.

[11] A. Schöbel. Set covering problems with consecutive ones property. Technical Report 2005-03, Georg-August Universität Göttingen, Institut für Numerische und Angewandte Mathematik, 2005.

[12] D. W. Wang and Y.-S. Kuo. A study on two geometric location problems. *Information Processing Letters*, 28(6):281–286, Aug 1988.

[13] E. Zitzler and S. Künzli. Indicator-based selection in multiobjective search. In X. Yao et al., editors, *PPSN*, volume 3242 of *Lecture Notes in Computer Science*, pages 832–842. Springer, 2004.

[14] E. Zitzler, M. Laumanns, and L. Thiele. SPEA2: Improving the strength pareto evolutionary algorithm. Technical Report 103, Swiss Federal Institute of Technology, 2001.

[15] E. Zitzler and L. Thiele. Multiobjective evolutionary algorithms: a comparative case study and the strength pareto approach. *IEEE Transactions on Evolutionary Computation*, 3(4):257–271, 1999.

[16] E. Zitzler, L. Thiele, M. Laumanns, C. M. Fonseca, and V. Grunert da Fonseca. Performance Assessment of Multiobjective Optimizers: An Analysis and Review. *IEEE Transactions on Evolutionary Computation*, 7(2):117–132, 2003.