Evolutionary Computation Methods for Instance Generation in Optimisation Domains

Alejandro Marrero Díaz

Supervisors: Prof. Eduardo Segredo
Prof. Coromoto León

Departamento de Ingeniería Informática y de Sistemas
Universidad de La Laguna

This dissertation is submitted for the degree of
Doctor of Philosophy

January 2024

The present thesis has been partially supported by Agencia Canaria de Investigación Innovación y Sociedad de la Información de la Consejería de Economía, Conocimiento y Empleo y por el Fondo Social Europeo (FSE) Programa Operativo Integrado de Canarias 2014-2020, Eje 3 Tema Prioritario 74 (85%) - under grant TESIS2020010005.

Acknowledgements

First of all, I would like to thank Eduardo Segredo and Coromoto León for their support and advice throughout the period which started with my Degree Thesis and culminated with the present PhD Thesis. Moreover, I would like to especially thank Emma Hart for her invaluable suggestions and collaboration during my research for this thesis. Her insights and recommendations have not only improved the overall quality of this thesis but also helped me to improve as a researcher.

Also, I would like to thank Jennifer Montesdeoca for accompanying me over the last few years and helping me see what really matters, and to my family, friends and colleagues for their support.

A very special appreciation goes to my girlfriend, Eva, for her love and encouragement throughout my PhD. Her unconditional support was a fundamental pillar during the ups and downs of this process.

Las escaleras se suben de frente,
pues hacia atrás o de costado
resultan particularmente incómodas.
(...)
Los primeros peldaños son siempre los más difíciles,
hasta adquirir la coordinación necesaria.
(...)
Cuídese especialmente de no levantar al mismo tiempo
el pie y el pie.

Julio Cortázar

Instrucciones para subir una escalera.

Abstract

The generation of instances of optimisation problems is a very common task in computer science. Traditionally, researchers apply statistical or pseudo-random methods to create instances with which to validate their proposals: algorithms or operators. At the same time, some authors have proposed sets known as benchmarks so that new proposals can be evaluated in these instances, and thus avoid the task of generating instances for researchers. However, these sets are often characterised by (1) being designed to be hard to solve by off-the-shelf, state-of-the-art algorithms at the time of their creation and (2) by their low diversity, meaning the instances tend to share many similar characteristics.

However, many of the proposals in the field of optimisation do not seek to evaluate state-of-the-art algorithms. Therefore, finding a solution to these benchmarks is not always the final objective in these publications. Hence, there is a need for instances that exhibit some diversity in their characteristics so that the strengths and weaknesses of a wider range of solvers can be evaluated. This factor is essential in problems such as Algorithm Selection; i.e., mapping a portfolio of algorithms to a set of instances. Since, in practice, there is no algorithm that can be expected to outperform others in every instance, collecting diverse instances with known best solvers could facilitate the evaluation of the strengths and weaknesses of the algorithms. Generating instances that are diverse from one another requires a method that (1) is capable of performing a space exploration and (2) has a mechanism for measuring diversity with respect to the rest of the instances encountered earlier in the search.

This thesis examines the problem of generating diverse and performance-biased instances from a portfolio of algorithms by proposing two major variants of Novelty Search (NS).The methods apply single- and multi-objective approaches to generate instances that are diverse and discriminatory, meaning they are designed to be diverse among themselves and also easy to solve for one target algorithm and not for others in a portfolio. By doing this, we aim to facilitate the generation of diverse sets of instances that can be used to fill currently existing gaps, perform algorithm selection within a portfolio and determine the regions of space where an algorithm excels/fails.

Although the proposals are mainly evaluated using the well-known Knapsack Problem (KP), experiments with the Travelling Salesman Problem (TSP) show that the methods can be generalised to other domains of combinatorial optimisation. The results suggest that both NS methods are able to generate diverse and discriminatory instances in the KP and TSP domains when using a portfolio of deterministic heuristics. Moreover, both methods outperformed previous Evolutionary Algorithm (EA) approaches in the KP domain.

Finally, the methods are integrated into DIGNEA, a Diverse Instance Generator with Novelty Search and Evolutionary Algorithms, a C++ framework that was developed during the research for this thesis to facilitate the generation of diverse and discriminatory instances for optimisation domains for the research community.

Contents

Nomenclature

The following list describes various symbols and initialisms that will be used later within the body of the document

Acronyms / Abbreviations

ASP Algorithm Selection Problem

BP Bin Packing Problem

CI Computational Intelligence

COP Constrained Optimisation Problem

CSP Constraint Satisfaction Problem

DOP Discrete Optimisation Problem

EA Evolutionary Algorithm

EC Evolutionary Computation

EIG Evolutionary Instance Generator

EMO Evolutionary Multi-objective Optimisation

EP Evolutionary Programming

ES Evolution Strategy

FOP Free Optimisation Problem

FSP Flow-shop Scheduling Problem

GA Genetic Algorithm

GP Genetic Programming

KP 0/1 Knapsack Problem

MDKP Multi-dimensional Knapsack Problem

ME Map-Elites

ML Machine Learning

MOIEG Multi-objective Evolutionary Instance Generator

MOKP Multi-objective Knapsack Problem

MOP Multi-objective Problem

MPP Menu Planning Problem

NKP Nonlinear Knapsack Problem

NN Neural Network

NPC Nondeterministic Polynomial-time Complete Problems (NP-Complete).

NS Novelty Search

NSGA-II Non-Dominated Sorted Genetic Algorithm II

OP Optimisation Problem

QD Quality Diversity

QKP Quadratic Knapsack Problem

SAT Boolean Satisfiability Problems

SOP Single-objective Problem

TSP Travelling Salesman Problem

Algorithm acronyms

NS_{lsf} Linear weighted single-objective Novelty Search Algorithm with feature-based descriptor

NS_{lsp} Linear weighted single-objective Novelty Search Algorithm with performance-based descriptor

NS_{ls} Linear weighted single-objective Novelty Search Algorithm

NS_{mof} Multi-objective-based Novelty Search Algorithm with feature-based descriptor

NS_{mop} Multi-objective-based Novelty Search Algorithm with performance-based descriptor

NS_{mo} Multi-objective-based Novelty Search Algorithm

Other symbols

A Set of all algorithms available for solving a problem P

c Combination of items that represents a solution for a Knapsack Problem instance

F Features or characteristics obtained from feature extraction techniques from P

m Performance metric to measure the performance of an algorithm with an instance

N Number of items in a Knapsack Problem instance

P Problem space of a potentially infinite set of instances for a problem domain

p_i Profit of the ith-item in a Knapsack Problem instance

ps Performance score obtained by an instance in the generation process

Q Capacity of a virtual knapsack in the Knapsack Problem domain

s Novelty score obtained by an instance in the generation process

w_i Weight of the ith-item in a Knapsack Problem instance

x Descriptor used by Novelty Search in the generation process

Y Performance space that maps each algorithm on A to an instance from P based on a performance metric m

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

Humans have a variety of ways of learning how to perform tasks. These include our own experimentation and learning from the teachings of a peer. In any case, throughout our lives, we learn that there is a wide spectrum of ways to perform a particular task. The choice of one method over another is often based on our beliefs, trends in our surroundings or our own experience. However, to consider only one way of performing a given task is a source of limitation. Even if we consider our methodology to be the most appropriate based on our experience, we may find ourselves in situations where time or resources do not allow us to proceed as we are used to. Therefore, it is important that we learn different ways of performing the same task that allow us to 'discriminate' those contexts or circumstances in which one method is more favourable than another. However, the lack of diverse scenarios is a problem when evaluating the different methods we can apply.

We can exemplify this statement in the following way. Suppose we have decided to go on a hike in the bush with our group of significant others. In this case, we are faced with the task of planning which items we want to take with us in our knapsack for this day in the countryside, and we do not have much time to do so. Each item has an associated weight (in kilograms) and a benefit that we have subjectively assigned to it. In addition, our knapsack has a capacity limit. Not only because it could break, but also because we would not be able to carry as much as we wanted on the route. In this case, our knapsack carries a maximum of 20 kilograms. Our goal is to carry with us the combination of items that give us the most benefit without exceeding the maximum capacity of the knapsack. Figure 1.1 shows a concrete example with a 20kg-capacity knapsack and the list of possible items we want to take with us.

Now we need to decide which items we are going to take and which we are not going to take. Similarly, let us assume that whenever we are faced with a similar problem,

Figure 1.1 Hiking Planning Problem

we always choose to plan in the same way; that is, we put the lightest items in the knapsack until we have filled it to its maximum capacity. And that is what we will do on this occasion as well, obtaining a total profit of 31.

On the other hand, there are other ways of doing this planning, such as, for example, considering introducing the most beneficial items first until the entire capacity of the knapsack is filled. The application of each methodology will give us a different plan for that day, and therefore a different total benefit. The main difference lies in the specific case we are 'solving', i.e., the different elements and the maximum capacity of the knapsack. However, if we had chosen the other approach, the *beneficial items first* method, we could have obtained a profit of 32. Figure 1.2 shows the possible solutions for this case.

Analysing the possibilities retrospectively, we see that our planning was not as good as it could have been given the time available, because we did not consider the characteristics of the elements involved (items and knapsack). The problem lies in the fact that we did not consider the possibility that different cases may work better for some methods than for others, and simply proceeded as we are used to from our experience. Also, the more experience we have with different cases and the more we can determine which methods work best for which specific cases, the better our planning

(a) Most beneficial items first

(b) Lightest items

Figure 1.2 Two different solutions for a Hiking Planning Problem. The efficiency of the method used to plan the day depends on the information of the instance.

will be. Something similar happens in the field of Optimisation within Computer Science.

Traditionally, efforts within the field of Optimisation have focused on the development of more powerful exact algorithms that yield optimal solutions for academic and real-world problems [136]. However, in practice, we find that it is not always possible to apply these methods due to their high demands for resources and computational time. Therefore, there are many lines of research and proposals to define algorithms that do not always obtain the optimal solutions to the problems in question, but that are capable of obtaining high-quality solutions with significantly fewer resources and time [13, 63].

One shortcoming that we find in the literature when evaluating these proposals is that the use cases, or 'instances', as they are commonly known in the field, have been created to be considerably hard to solve with the aim of evaluating state-of-the-art exact algorithms [51, 60, 95, 105, 106]. The main problem with these sets lies in the low levels of diversity. As a consequence, another problem arises, and that is the lack of information about which instances work best for the remaining algorithms.

Given a finite set of instances for an optimisation domain and a portfolio of algorithms, we find that the algorithms behave differently in different problem instances; no single algorithm dominates all others in every instance of the problem (i.e., performance complementary phenomenon). Although intuitively believed in the field before, this phenomenon was officially proposed in 1995 as the No Free Lunch Theorem (NFL) [31]. It is the task of the researcher to associate the appropriate algorithm to each instance of the set such that the total performance with the set is maximal.

This problem is known as the Algorithm Selection Problem (ASP) and was proposed by Rice in 1976 [112]. The early stages of the research in this thesis focused on the creation of a selector that maps each instance to the best-performing algorithm in order to efficiently solve the ASP in a specific domain. However, we soon realised the need to obtain information in terms of the performance of each algorithm with a set of instances. In this situation, we were able to determine that (1) the task of obtaining the performance of each algorithm with the instances is a very computationally expensive process and (2) the generation of sets of instances beforehand is absolutely necessary. Moreover, contrary to the trend in the field, these sets should be as diverse as possible in order to provide wide coverage of the performance spectrum. We thus find that within the context of ASP, the generation of instance sets plays a crucial role in the quality of the final selectors, which is why the efforts were shifted to this task: the generation of diverse and discriminatory instances.

In order to generate instances that are diverse from each other, a method is needed that (1) is capable of performing a space exploration and (2) has a mechanism for measuring diversity or novelty with respect to the rest of the instances previously seen during the search. The method employed in this thesis to accomplish this task is known as Novelty Search [79]. Proposed about a decade ago, Novelty Search belongs to a set of Evolutionary Algorithms called Quality Diversity (QD) approaches, which have accumulated a significant amount of attention in the fields of robotics, video game development, etc. To the best of my knowledge, Novelty Search methods have not been applied before to the generation of instances in optimisation domains. This method is characterised by focusing its search on novel solutions that have not been previously discovered and, at the end of its task, returning a set of high-quality solutions in a single run.

In particular, the present thesis focuses on investigating the problem of generating diverse and performance-biased instances for a portfolio of algorithms by applying two major variants of NS. Specifically, we seek to define methods that allow us to generate instances that are easy to solve for one algorithm and not for others. That is, given a portfolio of algorithms that we want to evaluate, our goal is to be able to generate sets of instances that are diverse among them and are defined to be solved by a target algorithm. In this way, the generation of instances also provides us with performance indicators for each algorithm in the portfolio. We thus aim to facilitate the generation of diverse sets of instances to fill currently existing gaps, perform algorithm selection within a portfolio, and determine the regions of space where an algorithm excels/fails. We present two variants of NS aimed at performing this task. The methods are fully

generalisable to any combinatorial optimisation domain. The proposals are evaluated mainly by using the well-known Knapsack Problem, an abstraction of the Hiking Planning example mentioned earlier. In addition, to illustrate that the methods can be generalised, they are also evaluated with the Travelling Salesman Problem.

The following section describes the main research questions that have been addressed during the research conducted in this thesis.

1.1 Research Questions

In this PhD thesis, we aim to explore the following research questions. It is important to note that the answer to the "to what extent" questions implies using specific qualitative and quantitative analyses to assess the quality of the final outcomes.

- **Question 1:** To what extent can a Novelty Search algorithm be used to generate diverse and discriminatory instances that aim to provide uniform coverage of the descriptor space for a portfolio of algorithms?
- **Question 2:** How diverse are the instances evolved for each target with respect to the performance-gap, i.e. the magnitude of the difference between the performance of the winning solver and the remaining solvers in the portfolio?
- **Question 3:** To what extent can a Novelty Search algorithm using either a feature or performance descriptor provide useful information of the opposite search space? Are the Novelty Search methods with a performance descriptor able to uniformly distribute the instances in the feature space and vice versa?
- **Question 4:** To what extent can the formulation of the instance generation problem impact the resulting sets of instances? In other words, are there substantial differences in diversity, space coverage and the total number of instances in the sets generated per run between a Novelty Search method using a single objective approach and a Multi-objective-based Novelty Search algorithm?
- **Question 5:** To what extent can Novelty Search methods be generalisable to other optimisation domains and portfolios when relying on performance-based descriptors?

1.2 Contributions and Overview

The research of this thesis presents two main contributions. Firstly, two NS approaches that are simultaneously capable of generating a set of instances that are diverse with respect to different search spaces (instance or performance) *and* that exhibit discriminatory but diverse performance with respect to a portfolio of solvers, where diversity, in this case, refers to variation in the magnitude of the performance gap. The approach is primarily evaluated in the KP domain to produce diverse and discriminatory instances for a portfolio of deterministic KP heuristics. Our results show that both novel approaches succeed not only in providing considerably better coverage for the instance and performance spaces, but also in obtaining larger sets of diverse and discriminatory instances in comparison to an approach that does not make use of novelty. Furthermore, in contrast to previous methods, such as that proposed by [2], a single run of our method returns a large set of instances that are diverse and discriminatory with respect to a single-target solver. It, therefore, needs to be run M times, where M is the number of solvers in the portfolio. In contrast, space-filling approaches such as those described by [2, 122], tend to converge to a single solution, meaning in the worst case scenario, they need to be run $i \times M$ times to generate i instances that are discriminatory for each of the M solvers.

The second major contribution is the development of a C++ framework called DIGNEA which makes it possible for other researchers in the field to apply these methods. The software is already available through a GitHub repository and contains a wide variety of solvers and several optimisation domains, such as the Knapsack Problem, the Travelling Salesman Problem and the Bin-Packing Problem.

The aforementioned research questions and contributions define the structure of the present thesis. The rest of the document is organised as follows. First, in Chapter 2, the different topics and areas related to this thesis are presented. Moreover, an overview of the methods and state-of-the-art for such subjects is given. Then, Chapter 3 is devoted to describing the proposal of two Novelty Search methods to tackle the problem of generating diverse and discriminatory instances in optimisation domains. In Chapter 4, both methods are examined in the KP domain. We perform several experimental evaluations to answer the majority of the research questions considered in this thesis. Chapter 5 presents an application of the methods to another optimisation domain: the Travelling Salesman Problem. The goal of the chapter is to illustrate how the method can be generalised to other domains with minimum effort by the user. Afterwards, in Chapter 6, we present DIGNEA, a Diverse Instance Generator with Novelty Search and Evolutionary Algorithms. This software has been created to allow the research

community to apply the methods described in Chapter 3. Finally, Chapter 7 provides a summary of the contributions and answers to the research questions, and provides a list of the publications resulting from the research of this thesis. It also briefly discusses some of the future lines of work of this research.

Chapter 2

Background

2.1 Evolutionary Computation

Evolutionary Computation (EC) is a research field within Computer Science. As readers might expect, EC is the idea of applying Darwin's Theory of Evolution to problem-solving [34]. Thus, the *natural evolution* metaphor — that of trial-and-error — and the survival of the fittest individuals in the population,[1] are the fundamental concepts of the basis of EC. The first reference to a nature-inspired search dates long before the computer breakthrough. In the 1940s, Alan Turing proposed a 'genetic or evolutionary search' for problem-solving in his famous paper *Intelligent Machinery* [126]. Almost two decades later, the first implementation and first experiment were successfully executed using computers [34].

Today, the field of EC is one of the three main branches in the foundations of Computational Intelligence (CI), the field centred on modelling biological and natural intelligent systems [35]. Even though the stochastic nature of EC techniques does not guarantee the optimal solution for the problem at hand, their ability to provide high-quality solutions in a reasonable time frame makes them highly popular these days in many different areas. They are applied in a wide range of scenarios where a search for high-quality solutions is required, such as pure Optimisation domains [21, 33, 89], Neural Network (NN) design in Machine Learning (ML) [83, 96], Robotics [127, 132] or Planning and Scheduling [48, 137] to name a few. Besides, there exist four main streams in the EC field: Evolutionary Programming (EP) [41], Genetic Algorithm (GA) [97], Evolution

[1] In practice, EC techniques do not always ensure the survival of the fittest individual in the population (neither does nature). The stochastic nature of algorithms and new techniques could provide other behaviours such as generational replacement.

Strategy (ES) [114], and Genetic Programming (GP) [74]. The term *Evolutionary Algorithm (EA)* is used to refer to the algorithms across subareas under the EC umbrella [34].

Figure 2.1 shows the general scheme of an EA flowchart. In the initialisation phase of the algorithm, a population of random solutions is created. Then, upon reaching the termination criteria (usually a number of generations or evaluations to perform), the algorithm proceeds as follows. First, random individuals are selected to create a mating pool of parents. Those individuals are mated by means of variation operators (recombination or crossover and mutation) to create a population of hopefully better individuals: the offspring. After that, the offspring and parent populations undergo a survivor selection mechanism to form the population of the next generation. When terminated, the algorithm usually returns the last population.

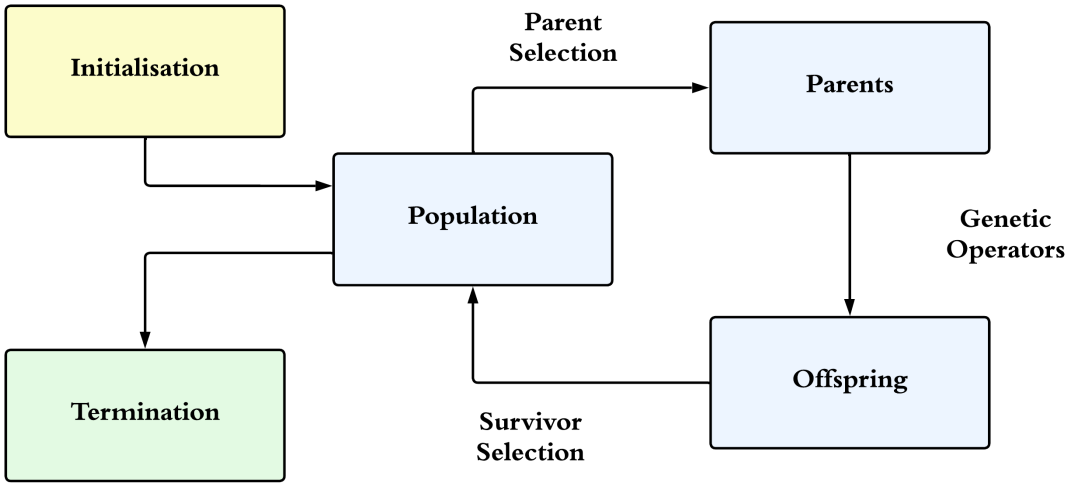


Figure 2.1 General scheme of an Evolutionary Algorithm flowchart

Alternatively, recent work reveals a new trend in the EC field: Quality-Diversity (QD) or illumination algorithms [19, 109]. In 2011, the first notion of QD algorithms was presented [79]. Lehman *et. al* questions the effectiveness of the traditional optimisation paradigm where EA algorithms measure the progress towards an objective function. In this paradigm, there exist a number of factors that are known to affect the performance of the algorithms. For instance, deception, such as local optima values, may prevent the algorithm from reaching the objective. In contrast, they proposed abandoning such a paradigm by instead rewarding solutions that present novel behaviour rather than high objectives scores. The authors named their proposal *Novelty Search*. Since their introduction around a decade ago, QD algorithms have provided a class of EA

methods which are now considerably prolific in the field of robotics, since they are capable of increasing the exploration of the search space at hand while simultaneously promoting higher quality solutions. Besides, they include the benefit of returning multiple high-quality solutions in a single run [25, 138].

There are two predominant classes of algorithms in the family of QD: Novelty Search (NS) [79] and MAP-Elites (ME) [99]. Both methodologies provide mechanisms for simultaneously enforcing the exploration of a search space while optimising for quality within each region. Even though most QD research is applied to the robotics and game development fields,[2] recently these algorithms are starting to filter into other optimisation domains (such as combinatorial optimisation) as a mechanism to expressly provide diverse solutions [129]. Moreover, the literature in the field suggests that defining suitable descriptors to depict the search space for both QD methods is not only critical and problematic, but can also have a significant influence on the dynamics of the algorithm [110].

In the present thesis, two NS algorithms are presented (see Chapter 3) to solve an optimisation problem: generating diverse and discriminatory instances for optimisation domains. Generating or collecting instances is a usual task that most researchers do when evaluating their algorithmic proposals for any optimisation domain. Even though the proposed algorithms are generalisable and can be applied to any optimisation domain, the principal optimisation domain addressed in this thesis is the well-known 0/1 KP.

2.2 The Knapsack Problem

The 0/1 Knapsack Problem (KP) is a well-known NP-Hard combinatorial optimisation problem with a broad range of applications in both academics and real-world situations. Studied since the 1890s, traditional lore suggests that the first reference to the term 'knapsack' was introduced by Tobias Dantzig [27].

Definition 1 *Given a virtual knapsack with a maximum capacity, a set of items T each one with an associated profit and weight values, find the combination c of items to include in the knapsack such that it maximises the total sum of profits without exceeding the maximum capacity of the knapsack.*

Formally, the KP is defined by Equation 2.1.

[2] See https://quality-diversity.github.io.

$$\begin{aligned} \max & \sum_{i=1}^{N} p_i c_i \\ \text{subject to: } & \sum_{i=1}^{N} w_i c_i \leqslant Q \\ & c_i \in \{0,1\}, \quad i \in N = \{1,\ldots,n\} \end{aligned} \tag{2.1}$$

The number of elements that can be introduced in the knapsack is given by N, p and w are the profits and weights of the items, respectively, and Q is the maximum capacity of the knapsack. Finally, the decision variables are represented by c.

Over the decades, multiple variants of the core KP have been proposed in academics [18]. The most relevant variations are:

- Quadratic Knapsack Problem (QKP) [44, 107]: Ever since its original proposal in 1980 by Gallo *et. al* [44], this particular variant of the 0/1 KP has garnered considerable attention within the academic community. Notably, the profitability of individual items is intricately intertwined with the selection of other items within a solution vector, denoted as x.

- Multi-objective Knapsack Problem (MOKP) [84]: The MOKP is a KP variant where more than one objective function is optimised simultaneously. This formulation allows the application of KP solvers and techniques to real-world problems, where sometimes several conflicting objectives or criteria must be optimised in parallel.

- Multi-dimensional Knapsack Problem (MDKP) [135]: Also known as the multi-constraint KP, the MDKP is a particular case of the 0/1 KP where the virtual knapsack is capable of storing items with two or more dimensions. Therefore, the capacity Q of the knapsack is defined as an M dimensional space with distinct capacities in each dimension.

- Nonlinear Knapsack Problem (NKP) [16, 58]: The NKP is a variant that incorporates non-linearity either in the objective function or constraints of the KP. The relevance of NKP derives from the fact that it can be encountered in diverse domains, either as a standalone problem or as a sub-problem. Additionally, the existence of non-linearity relations can present complexities and challenges, so traditional algorithms may not be directly appropriate to this variant.

Furthermore, there exist many real-world problems that derive from one or more KP variants. For instance, Menu Planning Problems [116], Project Management, Allocation,

Capital Budgeting, Cutting Stocks and many more problems [135]. Nevertheless, the focus of the present thesis is on the standard 0/1 KP.

From the algorithmic point of view, a substantial array of solvers for the KP across various classes have been proposed by the computer science and mathematics communities, i.e., from exact techniques such as Dynamic Programming [3, 22, 93, 113, 117] to heuristic [4, 14, 94, 135] and meta-heuristic algorithms like EAs [37, 75, 76, 89]. Even though there exist numerous proposals of exact solvers that are qualified to provide the optimal solution for KP instances, they are not the best option when solving large instances. As the size of the instances increases, the computational time required to reach the optimal solution is likely to increase dramatically. As a consequence, the field has witnessed the increased popularity of heuristics, meta-heuristics and other non-exact methods to obtain high-quality solutions in reasonable times [4, 14, 37, 76, 85, 89, 94, 135, 140]. However, different classes of algorithms (or even different configurations of the same core algorithm) may perform differently in certain instances. Depending on the resources available and the desired outcome, researchers must decide which algorithm best suits the particular needs of the instances at hand. This is not a straightforward task, and in fact, it requires considerable computation. Researchers have been facing this problem for decades, and the first formal definition of such a problem dates back to 1976, 'The Algorithm Selection Problem' [112].

2.3 The Algorithm Selection Problem

The No Free Lunch Theorem (NFL) [57, 134] established that the comparative performance of different black-box algorithms (the term black-box algorithm here refers to those solvers that do not include any problem- or instance-specific knowledge in their design to enhance the performance) across *all possible optimisation problems* is the same. This scenario has been observed for all NP-Hard optimisation problems such as Free Optimisation Problem (FOP) [39], also known as Unconstrained Optimisation Problem (UCP), Constraint Satisfaction Problem (CSP) [15] and Constrained Optimisation Problem (COP) [59, 72]. However, within every optimisation domain, there exist algorithms that can produce higher-quality results than others. Hence, it is well known that no single solver can excellently solve all instances from an optimisation domain; i.e., different algorithms may perform better than others for certain subsets of instances, necessitating the use of algorithm-portfolios which collectively provide high-quality coverage of the instance space of the problem. Thus, there is a practical

necessity to select the best-performing algorithm from a portfolio of solvers to solve instances of optimisation problems. Therefore, this leads to the per-instance Algorithm-Selection Problem (ASP) proposed by Rice in 1976 [112]. The formal definition of the per-instance ASP given by Rice is:

Definition 2 *Given a set of instances I of an optimisation problem P, a set of k algorithms $A = \alpha_1, ..., \alpha_k$ for solving P, and a metric $m : \alpha \times I \to \mathbb{R}$ that measures the performance of any algorithm $\alpha_i, i \in \{1, ..., k\} \in A$ on the instance set, construct a selector S that maps every problem instance $x \in I$ to an algorithm $\alpha_i, i \in \{1, ..., k\}$ such that the overall performance of S with I is optimal based on the metric m [112].*

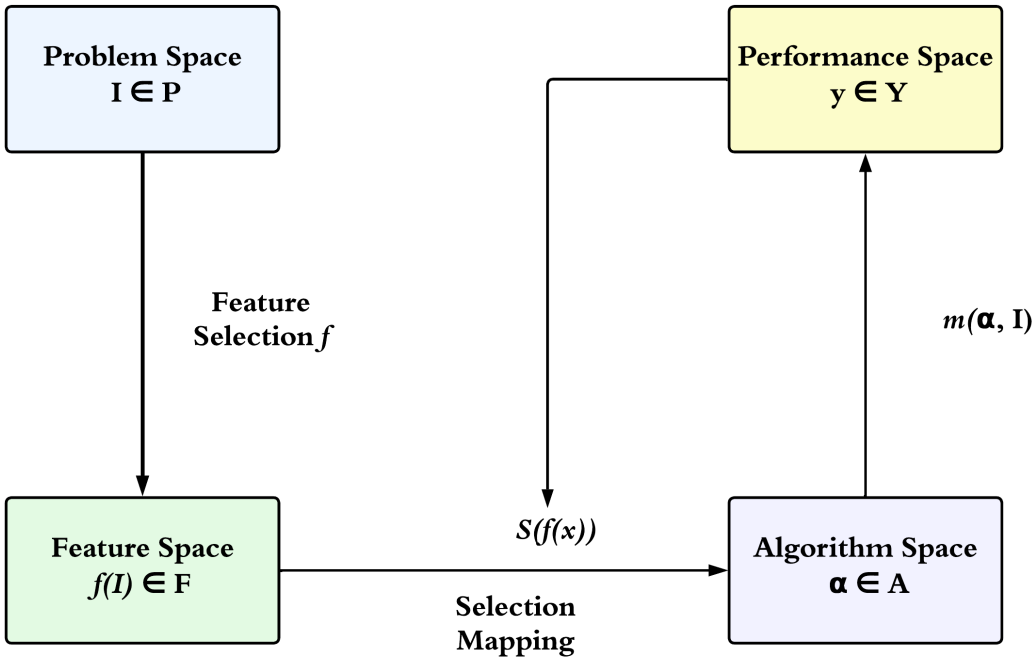


Figure 2.2 Scheme of the Algorithm Selection Problem as proposed by Rice in 1976.

Figure 2.2, shows an overview of the ASP as proposed by Rice where:

- P defines the problem space of a potentially infinite set of instances for the problem domain.
- The feature space F defines the set of features or characteristics obtained from P using feature-extraction techniques.

- The algorithm space A represents the set of all algorithms available for solving P.
- Y describes the performance space which maps each algorithm on A to an instance from P based on a performance metric m.

However, some observations may be made involving the above definition. First, the set of available algorithms A, could be replaced by a low-dimensional portfolio of diverse solvers, even different configurations of the same core solver. Moreover, the performance metric m can be highly sensitive to the optimisation domain at hand; i.e., cost functions, number of constraints violated for a solver, running time of the algorithm, etc. Therefore, a suitable metric m must be provided before addressing the ASP for a specific optimisation domain [118, 121].

Due to the rise of Machine Learning (ML) techniques and more powerful computational resources, the ASP has garnered considerable attention over recent years [70], with substantial work proposing approaches to predict either the performance of a given algorithm or the label of the best solver using large datasets of instances from the optimisation domain. For example, many algorithm selection approaches rely on training a machine learning algorithm to predict either the performance of a given algorithm or the label of the best solver, using a large set of representative instances from the domain for training [2, 54]. However, even though nowadays the *Selection Mapping* procedure in the ASP framework can be computed relatively straightforwardly thanks to those advantages, generating, characterising via feature selection, and solving instances with a portfolio of algorithms requires a significant amount of computational and research effort.

The present thesis is limited to the process of generating instances from the ASP framework. The purpose is to generate instances for optimisation domains that are not only diverse with respect to either the feature-space F or the performance-space Y instances, but that are also discriminatory with respect to the algorithms in the portfolio. Thus, the instance generation procedure can also label the instances with their corresponding best-performing algorithm in the portfolio A.

2.4 Instance Generation Methods

Traditionally, the literature related to instance generation in optimisation domains has mainly focused on generating as difficult as possible sets of instances with the aim of serving as benchmarks for existing or recently designed algorithmic techniques [26, 50, 51, 62, 65, 106, 130]. As a result, researchers were able to identify the strengths and weaknesses of algorithms to improve their performance. Such approaches, however, tend not to deal with the analysis of the instances and their diversity with respect to the instance-space, performing such an analysis only in terms of their hardness. Moreover, the degree of hardness or difficulty to solve an instance requires obtaining high-quality solutions, the optimal solution if possible. Besides, the traditional *hardness* evaluation of an instance relies on the elapsed time spent by a state-of-the-art algorithm to obtain a high-quality solution. However, this evaluation strategy involves heavy computational loads and long processing times, which would not be feasible to address in some cases. In fact, it is completely dependent on the algorithm selection, the implementation and the computational environment used to examine it. Nonetheless, some authors have proposed alternative metrics to evaluate the hardness of an instance [86].

In 2009, Smith-Miles *et al.* coined the term *instance-space* [123] to refer to a high-dimensional space that condenses a set of instances using a vector containing a finite list of measured features derived from the instance data. This feature-vector is commonly known as the *descriptor* of an instance. Projecting the descriptor onto a reduced lower-dimensional space (ideally 2D or 3D) provides researchers with an appropriate visualisation of the instance-space. Moreover, solver-performance metrics can be also superimposed as an extra indicator on the visualisation to reveal regions of the instance-space in which a potential algorithm outperforms other candidate solvers in the portfolio. As a result, instance-space visualisations can be useful in the field to understand areas in which instances are located based on different descriptors, areas of the space where solvers exhibit better performance metrics, or to assemble a portfolio of solvers.

Nevertheless, a vast majority of proposals tend to produce and evaluate benchmark instances only in terms of their hardness, i.e., how much time it takes for an exact solver to obtain the optimal solution in a certain computational environment.

2.4.1 Benchmark Instance Generation

Several authors have proposed a wide variety of methods to generate hard-to-solve instances in many optimisation domains. A number of studies on instance generation are based on drawing values from statistical distributions and selecting the top hardest-to-solve instances from the set [30, 106, 130]. On the other hand, few proposals are trying to move away from statistical methods to generate hard-to-solve instances; e.g., van Hemert *et. al* [131], have tried to harness the strengths of EC to generate hard-to-solve instances for the TSP domain. Most of the literature is related to TSP [26, 50, 51, 62] and other optimisation domains with direct application to real-world logistics and economics, such as Bin Packing (BP) [6, 30, 47, 115], Flow-shop Scheduling Problem (FSP) [130], Boolean Satisfiability Problems (SAT) [36, 60] or the KP [26, 65, 67, 106].

An example of the traditional approach based on statistical distributions is the study by Vallada *et. al* [130], where a new hard instance benchmark set for permutation FSP is proposed. The authors generated a total amount of 72,000 FSP instances from different parameter combinations using almost exclusively uniform random distributions. From that set, the top hardest-to-solve 240 small and 240 large instances were selected to create the benchmark set. They state that the experimental procedure took almost six years of combined CPU time to generate and solve the benchmark.

Other studies have argued that although the vast majority of combinatorial optimisation problems are known to be NP-hard, the practical hardness of the instances is usually not too difficult when created from random distributions. The work of Goerigk *et. al* is an example of this [50, 51]. While focusing on robust optimisation, they propose an optimisation model to generate hard instances for Min-Max optimisation combinatorial optimisation problems. Nevertheless, the authors claim that their approach is extendable to any combinatorial problem such as TSP or Selection. The approach is based on exact and heuristic methods to solve 'the optimisation problem of generating hard problem instances', and the instances generated for problems such as TSP or Selection are considered to be 500 times harder to solve by state-of-the-art Mixed-Integer Programming (MIP) solvers. Unfortunately, the authors did not evaluate the characteristic of the instances and exclusively considered the time-to-solve as the hardness of the instances.

Cárdenas-Montes [26], offers an alternative method to generate TSP instances that evaluates their hardness based on a set of metrics inferred from spatial attributes of previously solved samples. These features were then correlated with the hardness of the

instances. The hardness of the instances was deduced from the statistical distribution of the areas generated from the Delaunay triangulation, Dirichlet tessellation, and the distances between the cities fitted to a Weibull probability distribution. To harden the instances, a combination of a Linear Regression Model (LRM) and a Genetic Algorithm (GA) was created to exploit significant correlations between hardness and features.

Jooken *et. al* [65] recently proposed a new class of hard-instance problems for the 0-1 KP domain. They presented a stochastic multi-parameter generator that produces only one KP instance at a time. To evaluate the method, a dataset containing 3,420 hard-to-solve KP instances was generated to later solve them on a supercomputer using approximately 810 CPU hours. The instances were solved using state-of-the-art KP algorithms proposed by Pisinger: *Expknap* [104] *Minknap* [105] and *Combo* [93], proving that they were considerably harder than the 3,000 most difficult KP instances of the previous benchmark set [106]. Jooken defines the hardness of the instances as the required run-time of *Combo* to obtain the optimal solution. Furthermore, the authors indeed evaluated the diversity of the dataset and compared the results with work from in the field [120]. The results showed that, although the set was able to fill some gaps in the instance space, the instances were located in a small region of the space presenting low diversity behaviour. Thus, even though the work of Jooken *et. al* is primarily concentrated on generating a new state-of-art hard-to-solve KP benchmark, it differs from previous works in the field by analysing the location of the instances in a KP instance space.

Overall, it is important to note that centring our attention exclusively on creating increasingly hard-to-solve instances for exact state-of-the-art solvers may narrow the evolution of the field. Nowadays there exist many reasons why exact state-of-the-art solvers cannot be applied in many contexts: (1) it is impossible to afford the required time to obtain the optimal solution for a problem and it is necessary to obtain the highest quality possible solutions in a limited time, and (2) the end user may need more than one high-quality solution (without necessarily being the optimal one). For that reason, it is crucial to be able to rely on a portfolio of non-exact solvers that can yield high-quality solutions with reasonable usage of computational resources. As a result, research in the field might also pay attention to the conditions where such solvers excel/fail. However, evaluating the strengths and weaknesses of a solver is an arduous task that is often limited by the reduced diversity of instances that can be tested. As such, it is paramount to produce diverse sets of problem instances that also

exhibit some discriminatory behaviour, i.e., instances where a solver excels/fails with respect to other solvers in a portfolio of interest.

2.4.2 Discriminatory Instance Generation

Prior research in the field of instance-space analysis, which involves generating 2D visualisations of instance spaces, has shown that spaces constructed using commonly available benchmarks in the literature of well-known NP-Hard optimisation domains exhibit regions devoid of instances, and that certain solvers have limited coverage in the areas where they perform well. This has been demonstrated in studies by Smith-Miles *et. al* [119, 120, 122, 123]. Moreover, expanding such benchmark sets with real-world instances does not solve the problem that real-world instances tend to exhibit high levels of structure [55], and consequently, they tend to occupy small regions of the potential feature space.

In response to the aforementioned issue, recent research has focused on applying EC techniques to generate new instances to cover areas of an instance space currently lacking in instances [9, 119, 122], to generate instances that are easy or hard to solve for a particular algorithm, or to produce discriminatory instances with respect to a portfolio of solvers [11, 45, 46, 108]. Here the term *discriminatory instance* refers to an instance designed to be solved best by an algorithm in a portfolio of interest, providing a significant performance difference compared to said target solver and the remaining solvers. Overall, these studies allow researchers to provide new insights into the connections between the algorithm's performance and instance characteristics, thus enhancing the development of both better algorithm-selection and algorithm-configuration techniques. For instance, space-filling techniques, as proposed by Smith-Miles *et al.* [119, 122], have been employed to address the gaps in feature space in domains such as the TSP and Graph Colouring (GC). However, such approaches do not consider discriminatory behaviour in terms of algorithm performance over the instances. On the other hand, other research has centred attention on the evolution of new instances that demonstrate maximum discrimination with respect to a portfolio of solvers such as [2, 9, 108]. The main objective of those proposals is to maximise the performance difference between a *target solver*, i.e., a solver that is expected to obtain the best performance values, and other solvers in the portfolio in domains such as the Bin-Packing, TSP, and Knapsack respectively. Nonetheless, these methods tend to lack explicit mechanisms for generating instances that exhibit diversity with respect to the instance feature space.

In the TSP domain, Smith-Miles *et al.* [122] define a feature set to characterise the instances and propose a methodology to evaluate whether said set is sufficient to discriminate between instances and also between the performance of the different solvers in the portfolio. For that purpose, they describe an EA to evolve new random instances into easy (or hard) instances for a particular algorithm in the portfolio using an objective function that minimises (or maximises) the search effort required by a solver to produce a tour. The new instances are created to directly fill voids in a previously constructed instance space using benchmark instances. The benchmark set of instances is characterised by means of a feature-vector and then projected onto a 2D plane (e.g. using Principal Component Analysis (PCA) [42]), in which the projection is optimised to reveal the regions of strengths and weaknesses for several algorithms. This method is again demonstrated later for the Graph Colouring [119] domain, and most recently, in the domain of black-box continuous optimisation [101]. However, there is no guarantee the evolved instances will be discriminatory with respect to the portfolio.

Moreover, while Smith-Miles *et. al* focused their work on the TSP domain, Plata *et. al* [108] proposes a similar approach for the KP domain. Concretely, an EA method that generates KP instances which are designed to be easy or hard to solve for a specific algorithm in a portfolio of deterministic KP heuristics. The method uses a fitness function that maximises the performance gap between the target algorithm (the algorithm for which the instances are generated) and the remaining solvers in the portfolio. They generate two sets of instances for four different heuristics, easy-to-solve and hard-to-solve instances. After that, the instances are characterised by computing a 7D feature-vector to then analyse the feature distribution in the instance space. However, the main limitations of this approach are that: (1) no diversity metrics are given to quantitatively compare the set of instances, and (2) the representation of the easy and hard instances in the instance-space does not compare the set of instances generated for different solvers in the same figure, making it difficult to evaluate the space coverage and distribution among algorithms and classes of instances.

Alissa *et. al* [2] follow a similar methodology but applied to the domain of BP, i.e. a large set of instances are evolved to discriminate between four heuristic solvers. These approaches require running an EA multiple times to generate instances for a specific target algorithm. Besides, the number of distinct instances produced per run might vary, depending on the extent to which the final population has converged. The process must be repeated per each algorithm in the portfolio. Furthermore, these

approaches cannot be guaranteed to produce a diverse set of instances, since each run might converge to similar solutions.

To address the lack of diversity in the instance set, recent work attempts to introduce mechanisms to explicitly maintain diversity while evolving instances that are easy/hard for a target algorithm. The work of Gao *et. al* [45, 46] provides encouraging results for the TSP domain. The authors introduce a selection method in the EA evolution cycle designed to favour offspring instances that preserve the diversity in the population with respect to a desired feature, as long as the offspring instance presents a performance gap over a given threshold. Nonetheless, the study does not take into account different algorithms for the portfolio, which is reduced to only one algorithm: the 2-OPT heuristic [8, 102]. Moreover, even though the authors compared several Support Vector Machine (SVM) models to support their generation process, this presents another weakness of the study. In particular, there are two issues with the classification procedure: (1) the authors do not specify any train-validation-test division or cross-validation in the set of instances, which is commonly necessary for this kind of ML task to avoid overfitting[3] the data and (2) they only provide accuracy metric values for each classification task performed, which is not the preferred performance measure for classifiers [49, 61].

Additionally, Bossek *et. al* [9], also working on TSP, proposes a method to generate discriminatory instances that are also diverse with respect to a feature vector. The approach utilises novel mutation operators to encourage the exploration of the feature space using a simple iterative algorithm while still optimising for discrimination without explicitly preserving the diversity. Subsequently, the operators are also integrated into an EA method to optimise the generation of easy or hard instances for a target algorithm. Their results show that their method is capable of exposing a large difference in algorithm performance (easy for one solver and hard for its contender) while covering a wider spectrum of instance characteristics.

Alternatively, later work reveals there could be some benefits from the application of QD algorithms. In the field of instance generation, it was recently demonstrated that MAP-Elites could be utilised to evolve sets of TSP instances that are not only diverse with respect to a 2D feature vector, but that also exhibit discriminatory behaviour with respect to a portfolio of two TSP solvers [10]. A dynamically expanding population of TSP instances is mapped into cells of a 2D space. Each dimension of the 2D space refers to a user-defined feature derived from an instance, and each cell stores the

[3]Overfitting happens when the model is too complex relative to the amount of data and noisiness of the training data, and the model is restricted and does not generalise well with unseen data [49, 125].

instance with the best objective value found so far. The approach returns all instances contained in the map upon termination. Due to the application of MAP-Elites grid mapping, the method is able to discriminate between solvers and feature space. Bossek *et. al* claims that even though its proposal is rather simple, it could be generalisable to other optimisation domains. Nevertheless, the main drawback of the proposal is that the method fails to scale well as the number of features increases. Thus, the authors are constrained to define combinations of 2D feature-vectors to be explored independently from a set of over 150 potential candidate features to create the archive. Besides, regarding the objective values, the method is not target-dependent and the objective function is not explicitly defined, with the authors assuming a monotonically decreasing minimisation objective function.

In contrast, previous work related to the present thesis has demonstrated the use of Novelty Search in the KP domain [90]. NS is applied to generate instances that are not only diverse with respect to an 8D feature-vector, but also discriminatory to the performance of a portfolio of algorithms. A fixed-size population of instances is evolved and augmented with a dynamically growing archive. A feature vector is derived from each instance and used to calculate the novelty with respect to the current population and the archive. The method selects future generations using a linear weighted combination of objective fitness and novelty score. The method returns a *set* of diverse instances which are biased to the performance of a target solver. Concretely, the resulting datasets present the following characteristics: (1) cover a high proportion of the feature space relevant to a domain with quantitative and qualitative analyses to support the results; (2) contain instances in which the portfolio of solvers of interest demonstrate discriminatory performance; (3) contain instances that emphasise diversity in the *performance space*, i.e., they highlight a wide range of performance gaps between the target solver and the next best-performing solver in the portfolio. In fact, the last outcome of this work has rarely been addressed. The current trend in most studies is to evolve discriminatory instances while attempting to maximise the difference gap between a specific target solver and the next-best performing algorithm. Such approaches result in sets of instances which only emphasise the extremes of the regions of strength and weaknesses of the solvers. Nonetheless, it is undoubtedly essential to also locate the regions of the space in which one solver outperforms another, not just the extremes of their performances. Furthermore, the approach is evaluated in the KP domain using small instances and a portfolio of four different configurations of a parallel EA [89]. While the works present promising results, the method relies on calculating novelty as the distance between two feature vectors,

which is problematic for high dimensional vectors when distances become diminishingly small (the well-known curse of dimensionality [1, 49]). To the best of our knowledge, [9] and [90] are the only proposals to apply QD methods to the instance-generation problem with diversity and discriminatory requirements. It is clear that, although there is scope for improvement, QD methods are a promising approach to generate the problem of diverse and discriminatory instance generation.

In Chapter 3 of the present thesis, an in-depth description of two NS approaches (single and multi-objective variants) to generate instances is provided. There exists a substantial difference in whether to formulate an optimisation problem as a single-objective (SOP) or Multi-objective problem (MOP). In an MOP, two or more objectives, usually in conflict with each other, are optimised at the same time. Hence, a solution which increases the quality of one of those objectives tends to simultaneously decrease the quality of the others. Thus, the solution is a set of solutions representing the best trade-offs among objectives rather than a single optimal solution [33, 68, 69]. However, when a problem is formulated as an SOP, the goal is to increase the quality of only one objective. In this context, the solution to an SOP can be defined as a single solution. Moreover, there exist several strategies that are commonly used in the field to address an MOP as an SOP; e.g., the aggregation of the two or more objectives in a single linear-weighted function with a ϕ parameter defining how relevant each objective is in the evolution of solutions [28, 34].

2.5 Summary

Instance generation refers to the task of creating sets of problem instances that can be solved by some algorithmic technique. For every novel proposal, computer scientists are required to collect or generate by themselves problem instances to evaluate the strengths and weaknesses of their method. Although there exist benchmark sets of instances, they are designed with the objective of being difficult to solve for the top state-of-the-art solvers at the moment. They are thus lacking in diversity. Even though this is indeed an important research line in the computer science field, not all algorithms are designed to be state-of-the-art. Thus, evaluating a wide range of solvers with only sets of hard-to-solve and low-diversity instances prevents researchers from discovering where a solver excels or fails. Consequently, there is a need to generate diverse instances to complete existing sets and fill the regions of the space that remain unknown. Moreover, the discriminatory instances in terms of solver performance can

potentially help to illuminate which solvers may be preferable for certain areas of the space. This chapter reviewed the research related to the present thesis.

We will see in the following chapters that the research presented in this thesis differs from previous approaches in the instance generation field. While traditional approaches applied statistical and other mathematical methods to generate hard-to-solve instances, the methods presented here exploit the advantages of EC techniques to provide more complex generators. In contrast to conventional methods, the research is centred on the generation of diverse and discriminatory instances in optimisation domains. Thus, it not only allows researchers to create sets of instances that are biased to the performance of a target solver in a portfolio of interest, but that are also diverse with respect to a feature or performance space.

Finally, even though the problem of generating instances is introduced as a part of the ASP framework for algorithm selection, the work could be applied to pursue new research questions, for example, involving instance space representation, instance characterisation via hand-designed features, automated designed features, algorithm design, or even parameter tuning evaluation for algorithms across domains.

The remainder of this thesis is centred on addressing the research questions outlined in Chapter 1.

Chapter 3

Instance Generation Methods using Novelty Search

3.1 The Novelty Search Algorithm

NS is a flavour of EA proposed by Lehman *et. al* [79] with the aim of mitigating the problem of finding optimal solutions in deceptive landscapes. The core of NS is based on the following idea. Instead of seeking a goal by following an objective function in a standard evolutionary search process, the NS algorithm uses a function that rewards novel behaviour. Therefore, the candidate solutions are evaluated based on their diversity with respect to previous solutions rather than their 'fitness'. Although NS was proposed in robotics and control problems, such as maze navigation or biped walking tasks, many examples of the applications of NS now exist across many domains. Of particular relevance to this thesis is the work described by Buchanan *et. al* [17], in which NS is used to create a diverse set of robot morphologies—here there is no performance objective, the goal is exclusively to create diversity.

The NS algorithm works in a similar manner to a standard EA scheme (see Figure 2.1). Thus, the creation, selection and mating of individuals are conducted as we have detailed above in Section 2.1.

Once the offspring is created, NS begins to calculate the novelty for each candidate solution. Measuring *novelty* for a candidate solution requires the definition of an *instance descriptor* (x). A descriptor x is derived from the instance information and includes attributes that represent the instance. Then, NS uses the descriptor to calculate the novelty score or *sparseness* s for each candidate. The sparseness of a candidate is calculated using the current population, plus an external archive of previous individuals. Moreover, the K nearest neighbours and the Euclidean distance are commonly used

metrics to compare descriptors. The external archive of novel solutions is one of the fundamental components of NS [32]. Considered an unlimited-size set, it includes candidate solutions that were previously seen during the evolution. The literature on NS suggests different approaches on how to manage the external archive [53]; i.e., (a) include only candidate solutions which were novel at a certain point ($s >$ threshold) [79], (b) include the most novel candidates at each generation without considering candidates from the previous generations [82], (c) randomly populate it [78], or even (d) discard the archive altogether [100]. In the present thesis, the strategy selected is (a), i.e. when a candidate obtains a sparseness greater than a pre-defined threshold, it is included in an external archive of novel solutions.

After that, the evolution cycle continues as expected in an EA. The offspring and parent populations undergo a survivor selection mechanism to form the population of the next generation. When terminated, the algorithm usually returns the candidate solutions in the external archive. The general scheme of the NS using strategy (a) to operate the external archive is shown in Figure 3.1.

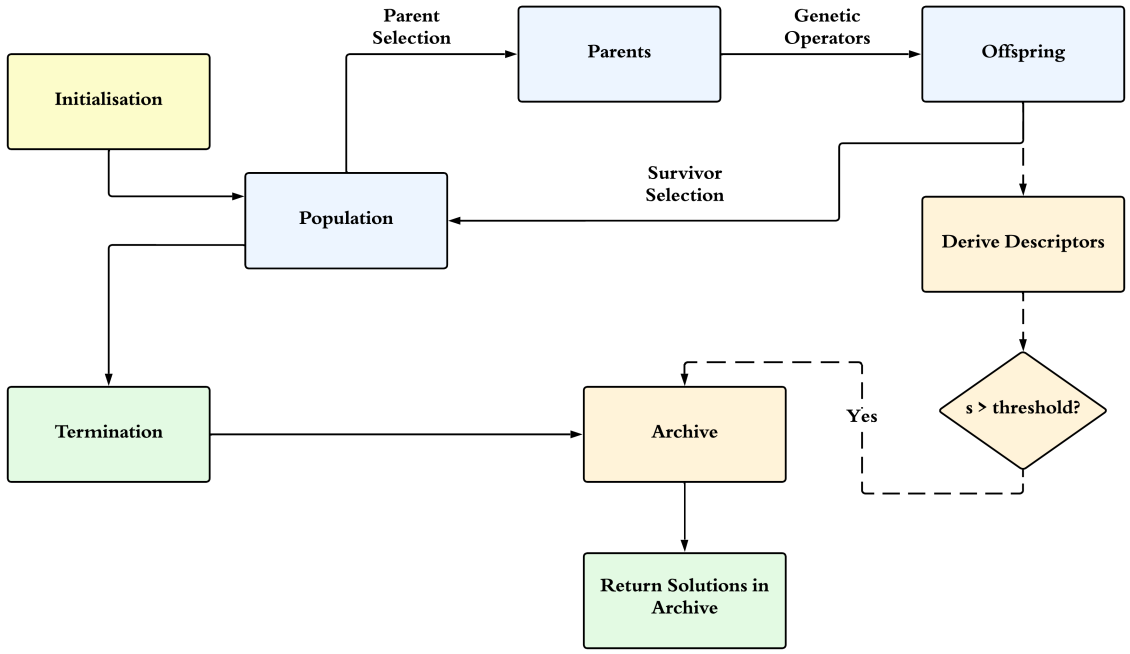


Figure 3.1 General scheme of a Novelty Search

In the present thesis, an NS method to discover diverse yet discriminatory instances in optimisation domains is proposed. The method is designed to create a set of instances that are diverse with respect to a descriptor and biased to the performance of a target solver. These instances can then be used to inform other problems such as algorithm-

selection, and instance space analysis. Whereas a straightforward NS approach would generate sets of diverse instances, they would not have the mandatory characteristic of being tailored to the performance of specific algorithms. It is important to note that to generate discriminatory instances for different solvers, the target algorithm must differ from one execution to another of the NS. In other words, although the method produces multiple instances in a single run, it must be run for each target solver in the portfolio. On the other hand, as mentioned in Section 2.4, pure evolutionary methods without any explicit diversity management mechanism would tend to generate sets of almost identical solutions [106, 108]. Hence, the NS presented in this thesis adopts a trade-off between searching for novel instances while also favouring higher performance score values. Two approaches are proposed to achieve the desired behaviour: a linear-weighted method based on that proposed in [24], and a multi-objective NS approach. To the best of our knowledge, this is the first time that a multi-objective-based NS is proposed for generating diverse and discriminatory instances in optimisation domains. Nevertheless, there exist some applications of multi-objective NS in other fields such as environment systems, evolutionary art and evolutionary robotics [5, 64, 98].

The remainder of this chapter will be devoted to defining how novelty is calculated with respect to descriptors, and to computing performance scores for the instances generated. Both NS approaches are defined in detail, and the general concepts of the methods are described, without any loss of generality, using the KP domain.

3.1.1 Calculating Novelty

The standard NS algorithm replaces the fitness calculation based on objective functions in a standard evolutionary algorithm with the calculation of a novelty score. To achieve this, it is necessary to define a descriptor to quantify how novel an individual is when compared to others in the population.

Given a descriptor x, usually a multi-dimensional vector with pertinent information on a solution, the most common approach in the field is to quantify the novelty of an individual by its *sparseness*. The sparseness of an individual measures the average distance between the individual's descriptor and its k-nearest neighbours [38, 52]. The k nearest-neighbours for an individual are determined by comparing the distance from the individual's descriptor to the descriptors of all other members of the population and to those individuals stored in the external *archive* of previous novel individuals. The sparseness s is then defined as shown in Equation 3.1:

$$s(x) = \frac{1}{k}\sum_{i=0}^{k} dist(x, \mu_i) \tag{3.1}$$

where μ_i is the descriptor of the ith-nearest neighbour of that individual whose descriptor is x, regarding a user-defined distance metric *dist*. The NS external archive is extended in two ways with each generation of the algorithm. On the one hand, we sample the population by randomly adding to the archive an individual with a probability of 1% (this is a common practice in the literature [124]). On the other hand, any individual in the current population with a sparseness s value larger than a user-pre-defined threshold t_a is also inserted into the archive.

Moreover, the NS complements the external archive with a separate list of individuals known as the *solution set*. The main difference between the external archive and the solution set is that the external archive is used to calculate the sparseness metric that drives evolution, and the solution set constitutes the final set of instances returned when the algorithm terminates [124]. The solution set is incrementally supplemented as the algorithm runs. At the end of each generation, each member of the current population is scored against the solution set by finding the distance to the nearest neighbour ($k = 1$) in the solution set. Those individuals that score above a particular threshold t_{ss} are added to the *solution set*. The solution set forms the output of the algorithm.

It is important to note that the solution set does not influence the evolutionary process by any means. On the contrary, this method ensures that each instance in the final set returned by NS has a descriptor that differs by at least the given threshold t_{ss} from the others. Lastly, there is no limitation in terms of the final size of either the archive or the solution set. In fact, either one can grow randomly in each generation depending on the diversity discovered by the current population. Figure 3.2 shows a flowchart of the NS algorithm which illustrates the difference between the archive and the solution set during the evolution.

Descriptor Representation for KP Instances

In order to calculate the *sparseness* (s) of an individual, we must define its descriptor, i.e. the representative vector of an instance being evolved. Here, the novelty of an instance can be defined either with respect to a set of pre-defined features of the instance (feature-based descriptor) or to the performance of the portfolio when solving that instance (performance-based descriptor).

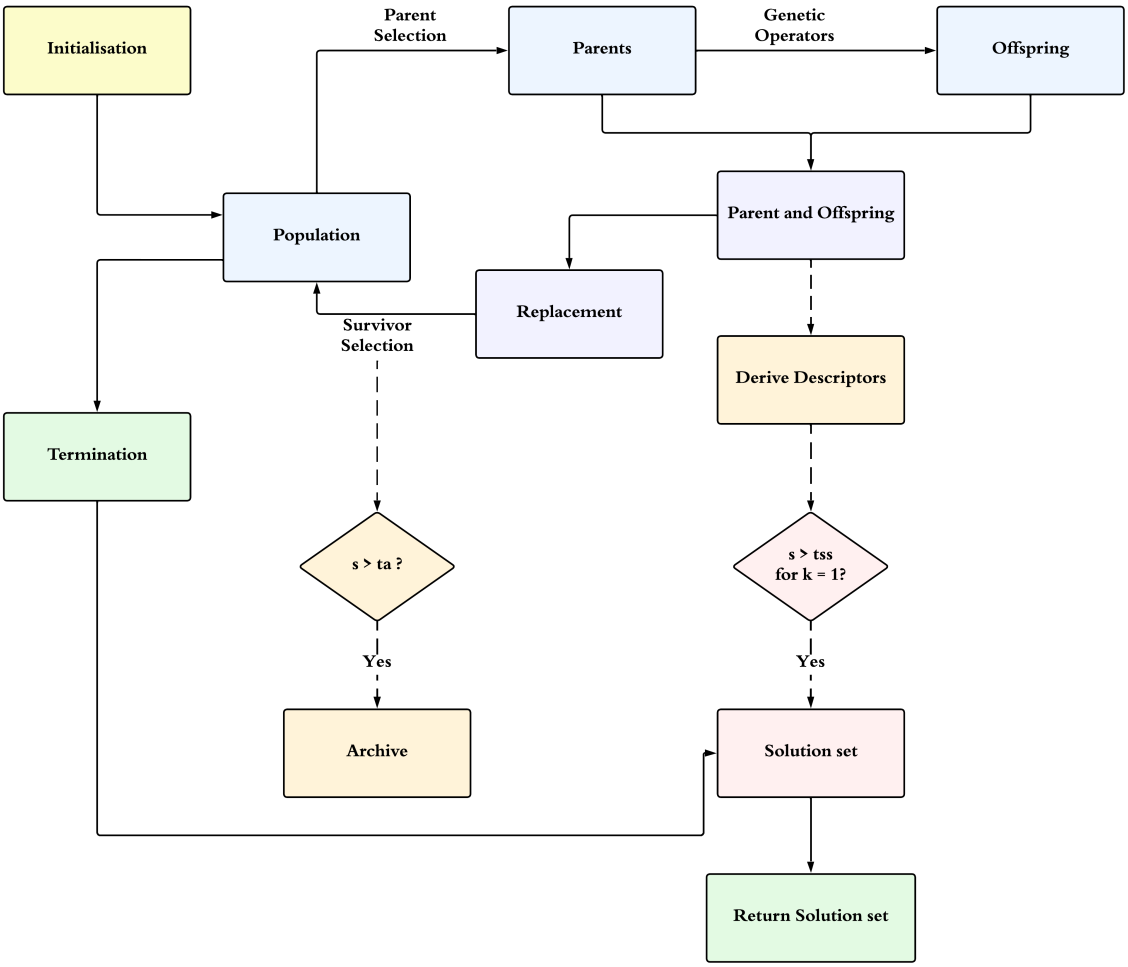


Figure 3.2 Novelty Search algorithm which includes a separate archive known as solution set.

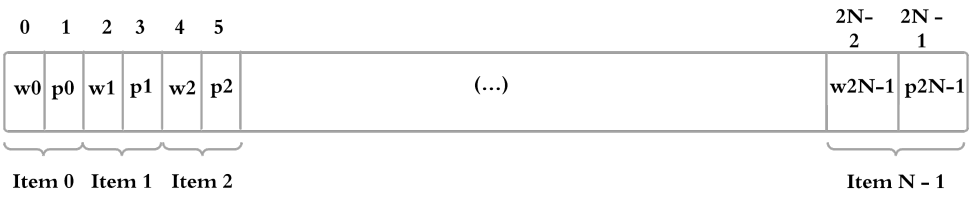


Figure 3.3 Representation of the instances as stored in memory (genotype).

A knapsack instance in NS is described by an array of integer numbers of size $N \times 2$ where N is the dimension (number of items) of the instance of the KP we want to create (see Figure 3.3), with the weights and profits of the items stored at the even and odd positions of the array, respectively. In the present thesis, the capacity C of the

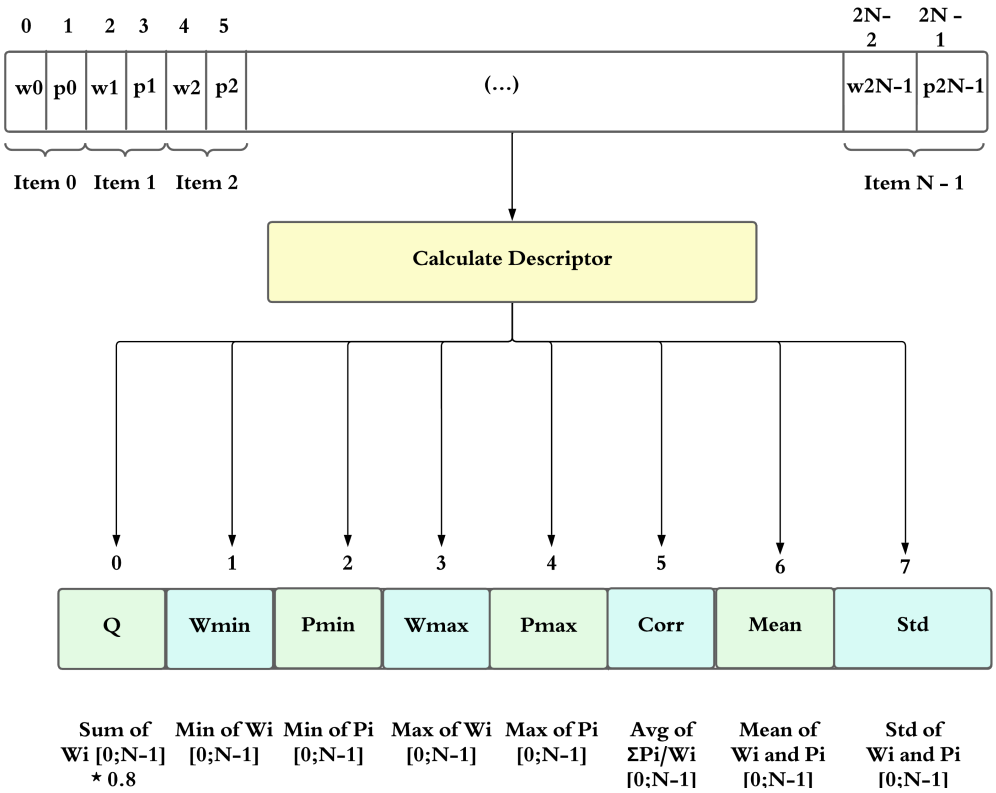


Figure 3.4 Representation of the creation of a feature-based descriptor from a KP instance genotype.

knapsack is determined for each new individual generated as 80% of the total sum of weights. Using a fixed capacity would tend to create insolvable instances if the weights of the instances increase significantly. However, this value could be included in the genotype so it can be evolved as well.

For the KP domain, the descriptor is built up with a set of eight features. Some of the features selected are inspired by those used by [108] containing: capacity of the knapsack; minimum weight and profit; maximum weight and profit; average item efficiency (also known as correlation); mean distribution of values between profits and weights ($N \times 2$ integer values representing the instance); the standard deviation of values between profits and weights. Figure 3.4 details how the feature-based descriptor is calculated from a KP instance.

On the other hand, a performance-based descriptor is defined as an M-dimensional vector with the average performance of each solver considered in a portfolio of size M. Considering a portfolio of M algorithms, then the novelty descriptor for an instance is calculated as an M-dimensional vector, where each element V_i represents the average

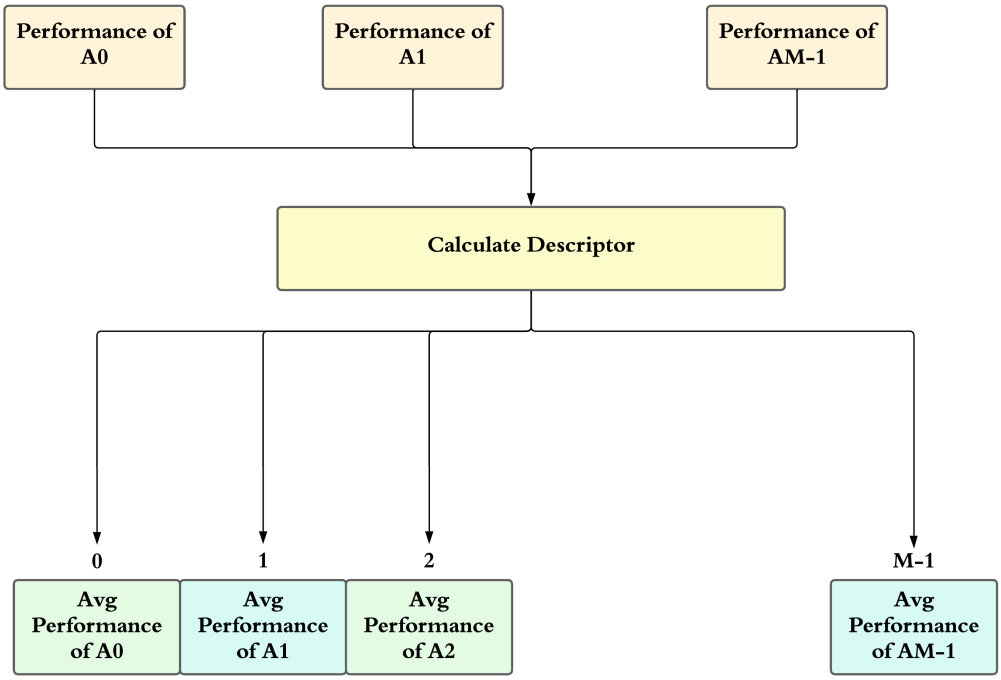


Figure 3.5 Representation of the creation of a performance-based descriptor from the performance of the algorithms in the portfolio.

performance over R repetitions of algorithm A_i on the instance. The main motivation behind the definition of a performance-based descriptor is to avoid defining a set of problem-dependent feature-based descriptors, which could be tedious to define and computationally expensive to calculate. Thus, the method could be applied to other domains where obtaining a feature-based descriptor might be a challenging task. Similarly, Figure 3.5 illustrates how the performance-based descriptor is derived from the performance of the algorithms in the portfolio for a specific instance.

Note that NS evaluates novelty with either descriptor given. This means the previous and following definitions are valid for both feature and performance descriptors. The main difference relies on the *space* where the novelty is searched for.

3.1.2 Calculating Performance

Whereas the *sparseness* (s) represents the novelty of an instance in the population, its *performance score* (ps) gives an insight into how well the target solver performs with this instance compared to the remaining algorithms in the portfolio. The reader should note the difference between the performance descriptor and ps. The performance descriptor detailed in the previous section refers to a search space in which NS searches

Algorithm 1: Evaluation Method

```
Input : current_pop, portfolio, archive, k, φ
for instance in current_pop do
    for algorithm in portfolio do
        apply algorithm to solve instance R times;
        calculate mean profit of algorithm
    end
end
calculate the novelty score(current_pop, archive, k) (Equation 3.1);
calculate the performance score(current_pop) (Equation 3.2);
calculate fitness(current_pop) (Equation 3.3);
return current_pop
```

for novelty, while ps is a performance metric used to calculate the fitness of instances. The performance score ps of an instance is calculated using Equation 3.2. For example, it can be computed as the difference between the mean profit achieved in R repetitions by the target solver, denoted as t_p, and the maximum of the mean profits achieved in R repetitions by the remaining approaches of the portfolio, defined as o_p. The number of repetitions R to perform depends on the nature of the algorithms in the portfolio; i.e., an evolutionary algorithm must perform several repetitions due to its stochastic nature, while deterministic heuristics only needs to be run once. In the particular case of the KP, profit is defined as the sum of the profits of the items included in the knapsack.

$$ps = t_p - max(o_p) \tag{3.2}$$

3.2 Instance Generation with a Linear-weighted Single-objective NS

The linear-weighted single-objective NS algorithm (NS_{ls}) follows a traditional EA approach for evolving instances [34]. First, NS_{ls} receives several parameters to construct the environment: D, a domain to create instances for; N, the dimension of the instance to generate, the number of neighbours k to calculate s for every instance, the value of $\phi \in [0, 1]$ for the linear-weighted computation of s and ps, the number of generations to perform and the portfolio of solvers to evaluate the instances. After that, both the external archive and the solution set, together with an initial population of randomly generated instances, are created and the algorithm starts the evolutionary process.

Algorithm 2: NS_{ls}

Input : D, N, k, ϕ, $generations$, $portfolio$

```
archive = ∅ ;
solution_set = ∅ ;
initialise(population, D, N);
evaluate(population, portfolio, archive, k, φ);
for i = 0 to generations do
    parents = select(population);
    offspring = reproduce(parents);
    offspring = evaluate(offspring, portfolio, archive, k, φ) (see Algorithm 1);
    population = update(population, offspring);
    archive = update_archive(population, archive);
    solution_set = update_ss(population, solution_set);
end
return solution_set
```

Until the maximum number of pre-defined generations is reached, NS_{ls} follows the traditional steps of an EA [34] with a few considerations. First, a *mating pool* of parent individuals is created by means of a random selection from the current population. The procedure defined to select the parents is *Binary Tournament Selection* [34]. Then, new individuals, i.e., offspring instances, are created based on individuals from the mating pool. This is known as *reproduction* in evolutionary computation. In the reproduction procedure of NS, two variation operators are applied: recombination (commonly known as crossover) and mutation [34]. By means of recombination, the information from two individuals in the mating pool is merged into two offspring individuals. Generally, recombination is a stochastic operator since the choice of what parts of the parents are combined involves some sort of randomness. In each generation of NS_{ls}, N offspring individuals are generated, thus ensuring $|offspring| = |population|$ in each generation of NS_{ls}. Mutation operators are applied individually to each new offspring to include a slight modification. In addition, mutation operators are always stochastic: the offspring depends on the outcomes of a series of random choices [33, 34]. The NS_{ls} algorithm presented here supports different flavours of mutation and recombination, and it is up to the researcher to decide which variant works best for each domain. However, in the present thesis, the choice of operators are *Uniform One Mutation* and *Uniform Crossover* [34]. It is important to note that since the representation of the instances in NS (see Figure 3.3) is based on the classical *vector-of-numbers representation*, the operators work in a similar manner to other optimisation contexts. There are no unusual cases because we are evolving instances rather than solutions for another optimisation problem.

When the variation operators are applied, the offspring instances are evaluated with the solvers' portfolio, their novelty s, performance score ps, and fitness f (see Algorithm 1). It is relevant to remark on the computational cost of the work performed in the evaluation stage of the instances. Not only must every instance be solved by each solver in the portfolio (even $R > 1$ times if the algorithm includes stochastic components), but s and ps are also calculated here.

The evaluation procedure requires several parameters: the current population of instances (the current population of instances is defined by the initial population at the start of the run and the offspring population in the main loop of the algorithm), the portfolio of solvers, the external archive and value k. As mentioned before, the evaluation procedure starts by solving every instance in the population with each algorithm in the portfolio R times. For each pair (instance, solver), the mean profit of the algorithm over the instance is calculated (see Section 3.1.2). The next stage in the evaluation requires computing scores s and ps for every instance in the current population. These values are calculated using Equation 3.1 for s and Equation 3.2 for ps. After that, both s and ps are combined into a *fitness* value f using a linear-weighted function where ϕ is the performance/novelty balance weighting factor (see Equation 3.3) [24]. NS_{ls} uses the parameter ϕ in the evolutionary process to obtain instances with certain trade-offs between their target performance ps and novelty s.

Higher ϕ values will tend to generate less diverse instances with a tight bias to the performance of the target solver. Note that defining $\phi = 1.0$ could lead NS_{ls} to work as a simple EA and converge the entire population of instances to similar solutions. On the other hand, lower ϕ values will emphasise the diversity aspect of the instances rather than their performance score. The evaluation procedure is detailed in Algorithm 1.

$$f = \phi * ps + (1 - \phi) * s \tag{3.3}$$

After evaluating the offspring, NS_{ls} undergoes the population update procedures. These procedures involve updating the current population, external archive, and solution set. Updating the population is the most straightforward process of the three. NS_{ls} performs an first-improve generational replacement where each individual in the population is replaced by its position-wise equivalent individual in the offspring population only if the fitness f is improved. Otherwise, the "older" individual remains. Next, NS_{ls} updates the external archive and solution set.

The external archive is populated with randomly selected samples of individuals from the current population and inserted into the archive with a probability of 1%,

a common practice in the literature [124]. Then, any individual from the current generation with a novelty score greater than a pre-defined threshold t_a is also included in the archive. Whereas the external archive can contain randomly selected and novel individuals, the solution set is populated exclusively with diverse solutions for which $s > t_s$ holds in each generation. The reasoning behind having both archives is that different levels of thresholds can be applied to each. For instance, having a less restrictive threshold for the external archive drives the evolution while requiring higher levels of diversity for the final set of solutions (see Section 3.1.1). Moreover, in NS_{ls} the score ps assigned to each instance is crucial. Thus, $ps > 0$ must be satisfied for any individual to be included in both archives. This restriction is only bypassed in the first generation of NS_{ls} to obtain some preliminary information to start the search for diversity. Finally, NS_{ls} returns the solution set. The pseudo-code of NS_{ls} is detailed in Algorithm 2.

3.3 Instance Generation with a Multi-objective NS

NS_{mo} considers the instance generation problem as a MOP. Whereas NS_{ls} combines the performance ps and novelty scores s into a single fitness value f by means of a linear-weighted function, NS_{mo} treats each objective separately. Thus, NS_{mo} discards the use of balance parameters, such as ϕ in NS_{ls}; instead, both objectives are optimised simultaneously. NS_{mo} is inspired by the one of the most successfully applied Evolutionary Multi-objective Optimisation (EMO) algorithms: the Non-Dominated Sorted Genetic Algorithm II (NSGA-II) [29]. Although this new evolutionary approach significantly differs from NS_{ls}, it maintains the identical domain representation presented in Section 3.1.1. Therefore, the same novelty descriptors and instance representation (genotype) is used for the KP domain in NS_{mo}.

The NS_{mo} algorithm starts in a similar manner to NS_{ls}, initialising the external archive and solution sets and then creating an initial population of random instances. After that, the initial instances are evaluated. The evaluation procedure for NS_{mo} is almost unchanged with respect to NS_{ls}, with the slight difference of avoiding the calculation of a fitness f value (see Algorithm 3). Both ps and s are calculated but not combined.

After the initial evaluation, the algorithm ranks the instances on different *fronts* based on non-dominance. The definition of domination considered in this thesis is the one proposed by Eiben [34]: given two solutions, both of which have scores based on some set of objective values, one solution is said to dominate the other if its score is at

Algorithm 3: Evaluation of Instances in NS_{mo}

```
Input : current_pop, portfolio, archive, k
for instance in current_pop do
    for algorithm in portfolio do
        apply algorithm to solve instance R times;
        calculate mean profit of algorithm
    end
end
calculate the novelty score(current_pop, archive, k) (Equation 3.1);
calculate the performance score(current_pop) (Equation 3.2);
return current_pop
```

least as high for all objectives, and is strictly higher for at least one for a maximisation problem. The domination is represented with the symbol $\succeq$. Formally, considering a maximisation problem, $A \succeq B$ (A dominates B) is defined as:

$$A \succeq B \Leftrightarrow \forall i \in \{1, ..., n\}\ a_i \geq b_i,\ and\ \exists i \in \{1, ..., n\},\ a_i > b_i \tag{3.4}$$

In scenarios where conflicting objectives exist, there is no single solution that can dominate all others, and a solution that is not dominated by any other will be known as *non-dominated* [34]. To rank the population on different fronts, NS_{mo} utilises one of the main components of NSGA-II: a fast non-dominated sorting operator [29]. The population of instances is classified on different fronts based on the non-dominance operator (see Equation 3.4); i.e, instances in F_1 are non-dominated individuals, F_2 includes the instances dominated by those in F_1, and so on. The algorithm compares every individual in the population to every other and for each individual i, it keeps two values: a dominance counter n_j and a set of solutions dominated S_j. The n_{p_j} determines the number of individuals that dominate individual j and therefore defines the front F_i that the individual belongs to; i.e., individuals scoring $n_p = 0$ belong to F_1, individuals with $n_p = 1$ are stored in F_2 and so on. Besides, the number of fronts F_i that the operator will return is not known beforehand. The fast non-dominated sorting algorithm is detailed in Algorithm 4.

After ranking the initial population, NS_{mo} begins the evolution of instances. A *mating pool* is created by means of random selection from the current population. The procedure to select the parents used in NS_{mo} is identical to NS_{ls}: *Binary Tournament Selection* [34]. Next, the individuals in the mating pool mate to create offspring individuals. While NS_{ls} will generate N offspring individuals, NS_{mo} includes twice as many individuals in the mating pool. Nevertheless, NS_{mo} does follow the same

Algorithm 4: *Fast Non-dominated Sorting* operator extracted from the original NSGA-II proposal [29]

```
   Input : population
 1 F = ∅;
 2 for p in population do
 3 |   S_p = ∅;
 4 |   n_p = 0;
 5 |   for q in population do
 6 |   |   if q ≽ p then
 7 |   |   |   n_p = n_p + 1 ;        Increment the domination counter of p
 8 |   |   else
 9 |   |   |   if p ≽ q then
10 |   |   |   |   S_p = S_p ∪ {q} ; Add q to the set of solutions dominated
   |   |   |   |    by p
11 |   |   |   end
12 |   |   end
13 |   end
14 |   if n_p = 0 then
15 |   |   p_rank = 1 ;                    p belongs to the first front F_1
16 |   |   F_1 = F_1 ∪ {p}
17 |   end
18 end
19 i = 1;
20 while F_i ≠ ∅ do
21 |   Q = ∅;
22 |   for p in F_i do
23 |   |   for q in S_p do
24 |   |   |   n_q = n_q − 1;
25 |   |   |   if n_q = 0 then
26 |   |   |   |   q_rank = i + 1 ;          q belongs to the next front F_{i+1}
27 |   |   |   |   Q = Q ∪ {q};
28 |   |   |   end
29 |   |   end
30 |   end
31 |   i = i + 1;
32 |   F_i = Q;
33 end
34 return F
```

approach as NS_{ls} in the reproduction stage of the evolutionary process. This means that recombination and mutation operators are applied to each individual offspring. Moreover, the same operators are used for NS_{mo}; i.e., *Uniform One Mutation* and *Uniform Mutation* [34].

Algorithm 5: NS_{mo}

Input : D, N, k, *generations*, $portfolio$

```
archive = ∅ ;
solution_set = ∅ ;
initialise(population, D, N);
evaluate(population, portfolio, archive, k);
rank_population(population);
for i = 0 to generations do
    parents = select(population);
    offspring = reproduce(parents);
    offspring = evaluate(offspring, portfolio, archive, k) (see Algorithm 3);
    population_c = {population ∪ offspring};
    population = update(population_c);
    archive = update_archive(population, archive);
    solution_set = update_ss(population, solution_set);
end
return solution_set
```

Furthermore, once the offspring population is created and the variation operators are applied, the offspring individuals are evaluated using Algorithm 3.

After that, the offspring and current populations are combined into a population $population_c$ of size $|population_c| = 3N$. The $population_c$ is then used to update the current population. Note that the idea behind combining both populations is to ensure the preservation of elitism during the entire process [29, 34]. In order to update the population, NS_{mo} begins by applying the non-dominated sorting operator to classify the individuals in different fronts and prioritise those instances which are non-dominated (see Algorithm 4). Furthermore, the update procedure involves another key component of NSGA-II: the *Crowding Comparison operator* [29]. This operator is used to estimate the density of neighbour solutions around an individual in the population [29, 34]. So as to calculate the crowding distance to an individual i, each front has to be arranged in ascending order for each objective function value. Then, for each objective function, the distance is assigned as the normalised difference between the objective values of two adjacent individuals in the front [29]. It is important to note that individuals in the bounds of each objective function are assigned a distance value of ∞. For instance, if $F_i = x_0, x_1, ..., x_m$ is a front already sorted in ascending order based on a certain objective function, x_0 and x_m will receive a distance value equal to ∞. Once all the objective functions have been evaluated, the overall crowding distance results as the sum of the distances for every objective function [29]. Finally, the front is sorted based on the overall crowding distance value of each individual

inside. Afterwards, the population is updated with the first N individuals found in F. Starting with the best non-dominated individuals in F_1, if the size of the front is smaller than N, all individuals are selected as members of the new population. Then, the population is completed with individuals from subsequent non-dominated fronts (i.e., F_2, then F_3, etc.) based on their crowding distance. NS_{mo} continues the process until N individuals are filled in the new population.

Next, NS_{mo} updates the external archive and solution set identically as NS_{ls} does. For instance, the external archive is populated with randomly selected samples of individuals from the current population and inserted in the archive with a probability of 1%, and any individual from the current generation with a novelty score greater than a pre-defined threshold t_a is also included in the archive. The solution set is populated only with diverse instances that score $s > t_{ss}$ and $ps > 0$. Finally, although NS_{mo} works as a multi-objective EA, it does return a *solution set*, similar to the way NS_{ls} operates, to avoid discarding solutions that are diverse but do not belong to a proper Pareto Front. In this way, NS_{mo} not only treats the instance generation problem as a multi-objective problem; it also benefits from the use of archives. The pseudo-code of NS_{mo} is detailed in Algorithm 5.

3.4 Summary

In this chapter, two main NS methods for generating diverse and discriminatory instances with respect to a portfolio of solvers in optimisation domains are proposed.

On the one hand, an NS approach based on [90] known as NS_{ls} solves the problem of instance generation by means of a weighted combination of performance and diversity to drive the evolution of instances. Additionally, a novel NS method where the performance and diversity are optimised simultaneously is proposed as well: NS_{mo}. The method is based on the NS_{ls} plus the well-known NSGA-II algorithm for multi-objective optimisation. Moreover, both methods are generalisable in terms of the space used to seek diversity: a feature space based on domain-dependent descriptors, or a performance space with respect to the portfolio of interest. Hence, combining both methods and spaces to search for diversity yields the four different approaches presented in this chapter. Whereas deriving feature descriptors could lead to a deeper understanding of the instances, performance-based descriptors do not require the definition of a feature-based descriptor to characterise instances, which is a clear advantage in domains where there are no intuitive features, or where calculating features is computationally expensive. Additionally, it also facilitates the generalisation

of the method to other domains since it does not require an intensive problem domain analysis and is a much more straightforward method in comparison to the feature-based approach. An example of the above is presented for the TSP domain in Chapter 5.

Finally, of the main advantages of the methods proposed is that a *set* of diverse and discriminatory instances is returned in a single run. By contrast, existing methods in the literature, as stated in Chapter 2, such as [2, 108, 122], need to be run repeatedly to generate multiple instances since the EAs often converge to a single solution; furthermore, there is no guarantee that repeated runs will deliver unique solutions.

The next chapter presents an in-depth evaluation of the methods in the KP domain.

Chapter 4

Experimental evaluation

Hofstadter's Law: It always takes longer than you expect, even when you take into account Hofstadter's Law.

Douglas Hofstadter
'Gödel, Escher, Bach: An Eternal Golden Braid'

This chapter details the experimental evaluation performed for the KP domain in the present thesis. The main motivation for the experimental assessment is to evaluate the extent to which both novelty search approaches, NS_{ls} and NS_{mo}, can be used to generate diverse but discriminatory instances for the KP problem with respect to a portfolio of solvers. Moreover, each NS algorithm is evaluated on both previously detailed descriptors: feature-based and performance-based (see Section 3.1.1). Thus, we can evaluate the quantitative and qualitative diversity of the instances from the approaches not only in the traditional instance feature space, but also when disregarding them.

To this end, both methods are evaluated using a portfolio containing a set of simple, deterministic, and KP-dependent heuristics [108]. These are well-known solvers commonly used in the optimisation field, namely: Default (Def), which selects the first item available to be inserted into the knapsack; Max Profit (MaP), which sorts the items by profit and selects those items with largest profit first; Max Profit per Weight (MPW), which sorts the items by their efficiency (ratio between the profit and weight of each item) and selects those items with the largest ratio first; and Min Weight (MiW), which selects items with the lowest weight first. Although these

algorithms are very straightforward to implement, a pseudo-code is provided for each one in Appendix A.

This experimental evaluation addresses the first four major research questions of this thesis from Section 1.1:

The chapter is organised as follows. First, the experiments related to NS_{ls} are presented, where Questions 1, 2, and 3 (see Section 1.1) are answered for both descriptors: feature- and performance-based. In order to answer Question 3, a traditional evolutionary algorithm is used as a base method to compare against NS_{ls}. After that, the same questions are addressed for NS_{mo}. In this scenario, the well-known NSGA-II algorithm is used as a base algorithm to answer Question 3. Then, in order to answer the fourth question of this thesis, an in-depth comparison between NS_{ls} and NS_{mo} is presented. This section compares both methods for each descriptor and emphasises the possible benefits of an ensemble approach between the two algorithms.

Table 4.1 Parameter settings for NS_{ls}, which evolves the diverse population of discriminatory instances.

Parameter	**Value**
Knapsack items (N)	50
Weight and profit upper bound	1,000
Weight and profit lower bound	1
Population size	10
Crossover rate	0.8
Mutation rate	1 / ($N \times 2$)
Generations	1,000
Portfolio	Default, MaP, MPW, MiW
Repetitions (R)	1
Distance metric	Euclidean Distance
Neighbourhood size (k)	3
ϕ	0.85
Thresholds (t_a, t_{ss})	3.0

4.1 NS_{ls} Experiments

First, NS_{ls} is run considering a portfolio of deterministic heuristics and both descriptors (features and performance). Due to the stochastic nature of the algorithm, NS_{ls} was run 30 times for each solver and then the results were combined. The results are two datasets of instances generated by NS_{ls} to favour a specific solver whose diversity is defined using the descriptors detailed in Section 3.1.1, i.e., a feature-space descriptor for NS_{lsf} and a performance-space descriptor for the NS_{lsp} approach. The parameter setting for all experiments involving NS_{ls} are detailed in Table 4.1. Moreover, the values for each parameter are defined after considerable computational work performed in parameter tuning experiments [90].

4.1.1 NS_{ls} and Feature-based Descriptor

We denote NS_{lsf} as the configuration of NS_{ls} that is run to generate KP instances using a feature-based descriptor to search for diversity. The approach is executed 30 times for each of the four targets in the portfolio and then the instances are combined into a single dataset. After that, the instance information[1] plus the feature-descriptor is reduced to 2D for better visualisation. The procedure of dimensionality reduction is performed by means of standardisation, followed by the application of Principal Component Analysis (PCA) [42]. The data are standardised by removing the mean and scaling to the unit variance. For instance, the standard score of a sample x consisting of the feature-descriptor and the instance data is calculated as:

$$z = (x - \mu)/s \tag{4.1}$$

The results are shown in Figure 4.1. It is evident that NS_{lsf} was able to generate four different clusters of instances, one for each solver in the portfolio. Even though the number of instances generated per target varies from solver to solver (720 instances for Default, 220 for MPW, 2260 for MaP, and 2090 MiW), the ratio of unique instances remains stable at around 30 per cent of the total amount (see Table 4.3 for more details) after 30 repetitions of the method for each solver. The difference in terms of instances generated per solver was determined by (1) the stochastic nature of the method, (2) the idiosyncrasy of each solver (making it easier or harder to generate instances biased to their performance), (3) the need for specific NS threshold adjustment (t_{ss}, t_a) when

[1]The term instance information refers to the attributes that define a KP instance, such as the capacity (Q), and the profits and weights of each item in the instance.

dealing with hard-to-generate-for solvers, or (4) possible limitations related to tackling the instance generation problem as a single objective problem.

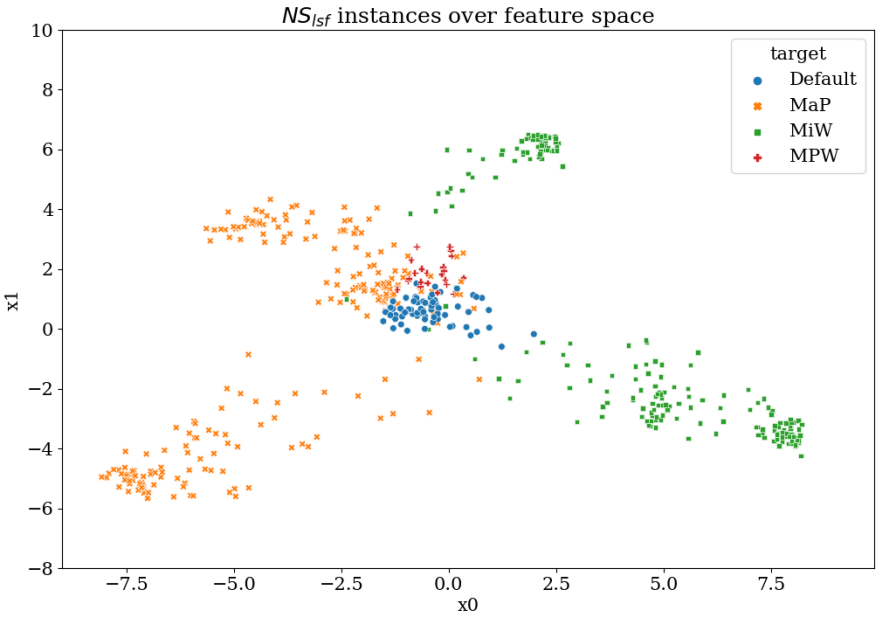
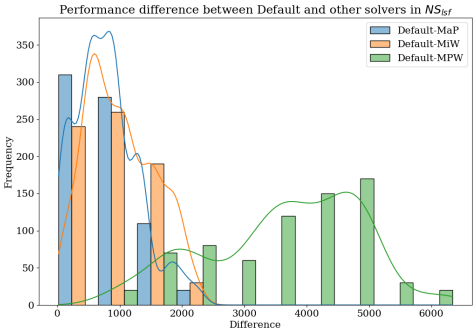
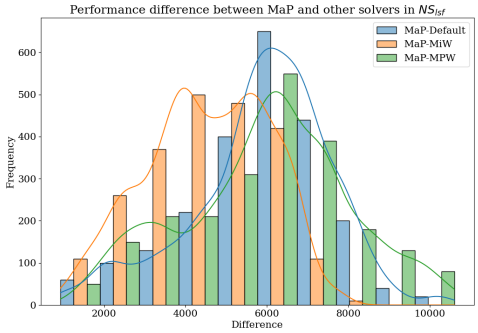


Figure 4.1 PCA is applied to a dataset containing the feature descriptors plus the instance information of all the instances generated by NS_{ls} when using a feature-based descriptor to search for novelty. Blue points are the instances generated for Default, orange crosses for MaP, green squares for MiW, and red pluses for MPW.

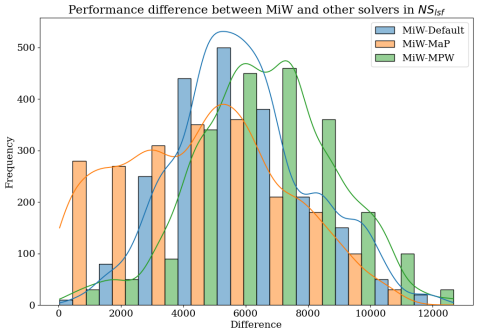
Moreover, a quantitative evaluation of the space coverage from the generated instances is provided by means of the exploration uniformity (U) metric [53, 77]. This procedure permits a fair comparison of the distribution of the instances with a hypothetical Uniform Distribution (UD) in the same feature space. The procedure starts by dividing the environment into a grid of 25×25 cells, after which the number of instances in each cell is counted. After that, the Jensen-Shannon divergence (JSD) [43] is applied to compare the distance of the distribution of instances with the ideal UD. The U metric is then calculated using Equation 4.2, where δ denotes a $2D\text{-}descriptor$ associated with an instance. This descriptor is defined as the two principal components of each solution extracted after applying PCA to the combination of the instance information plus the feature-descriptor, as detailed above. Moreover, the higher the U score, the better. Thus, obtaining a score of 1 proves a perfectly uniform distributed set of instances.



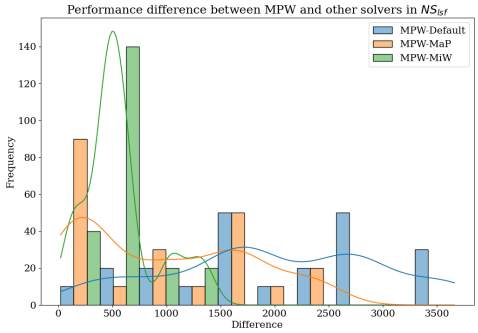
(a) Default



(b) MaP



(c) MiW



(d) MPW

Figure 4.2 Distribution of the performance gap between the Default, MaP, MiW, and MPW approaches and other solvers in the portfolio by considering the instances generated for the former when running NS_{lsf}. The X-axis scale varies from one sub-figure to another to produce better visualisations of the bars in the plots. Therefore, the differences between algorithms depend on the solver that is taken as a reference.

$$U(\delta) = 1.0 - JSD(P_\delta, UD) \tag{4.2}$$

The set of instances generated by NS_{lsf} scores $U = 0.5883$. It is crucial to note that the U score is highly related not only to the number of instances generated, but also, and even more importantly, to the ϕ factor used in NS_{ls}. When generating instances with NS_{ls}, ϕ determines the importance of the performance and novelty scores in the fitness of the instances (see Section 3.2). Lower ϕ values could tend to generate datasets of instances considerably more spread out across the space since the importance of the novelty score will be reinforced (see Equation 3.3). An in-depth evaluation of the impact of ϕ in NS_{ls} is discussed in [90]. Section 5.2.1 replicates the analysis for the TSP domain.

Next, a quantitative analysis of the performance-gap is presented to answer question number three of this experimental evaluation for NS_{lsf}. This is done by relying on the instances generated for each solver by NS_{lsf} and the spread in the magnitude of the performance gap as defined in Equation 3.2. Although NS_{lsf} is not designed to generate diversity in terms of this metric, the results prove that while a significant number of instances have a relatively small gap between the performance of their target and other solvers, it is possible to find instances that exhibit performance diversity as well. The distributions of performance gap between instances are shown in Figure 4.2. The x-axis represents the magnitude of the difference between the target solver and the other (performance-gap), and the y-axis is the frequency with which the value appears; i.e, the number of instances that exhibit a performance gap.

The distributions for every heuristic exhibit some multi-modal behaviour, and even though the scores for a few instances reveal a performance-gap that is almost negligible (close to zero), a large number of other instances obtain much higher scores (> 6000) for the MaP and MiW solvers. Moreover, even though the low number of instances obtained for Default and MPW could have affected the distribution when compared to MaP and MiW, some diversity is evident as well. The fact that there exists diversity in terms of the performance gap indicates that it is possible to train ML classifiers to make use of the features of an instance to rank the solver that would obtain the best performance metrics in a portfolio of interest.

4.1.2 NS_{ls} and Performance-based Descriptor

NS_{lsp} is the abbreviation for the configuration of NS_{ls} when running to generate KP instances that are diverse in the performance space, i.e., using a performance-based descriptor to search for diversity. Similar to the previous evaluation, the method is run 30 times for each of the four targets in the portfolio and then the instances are combined into a single dataset. The parameter setting for NS_{lsp} is identical to NS_{lsf}, as detailed in Table 4.1.

In addition, once the experiments are completed, an analogous dimensionality reduction procedure is applied to the resulting dataset. Thus, standardisation and PCA are applied to the dataset, with the slight difference that, in this scenario, the feature-based descriptor is exchanged with the performance-based descriptor.

Since NS_{lsp} is designed to search for novelty in the performance space, it is more convenient to evaluate the results in this space. Figure 4.3 illustrates the results, which show that NS_{lsp} was also able to generate four different clusters of instances, one for each solver in the portfolio. Although the instances follow a pattern similar to

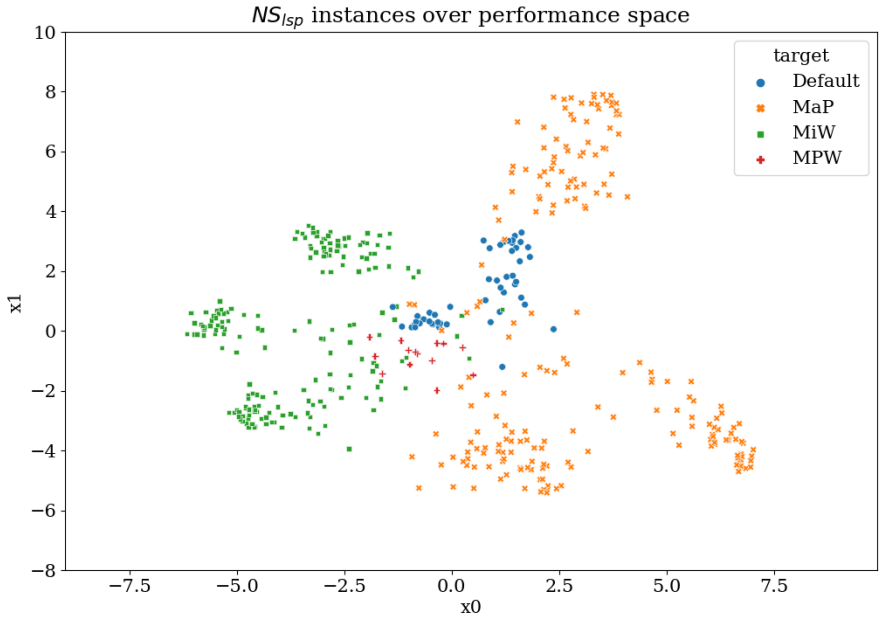


Figure 4.3 PCA is applied to a dataset containing the performance descriptors plus the instance information of all the instances generated by NS_{ls} when using a performance-based descriptor to search for novelty. Blue points are the instances generated for Default, orange crosses for MaP, green squares for MiW, and red pluses MPW.

Figure 4.1, note that in Figure 4.3 the instances are plotted over the performance space rather than the feature space.

The space coverage was also calculated for NS_{lsp}, yielding a score $U = 0.6426$. At this point, the NS_{lsp} U metric is not comparable with previous results from NS_{lsf} since the values are obtained in different spaces. However, since NS_{lsf} provide diversity in the performance space without explicitly requiring it, it is rather interesting to evaluate both approaches, NS_{lsf} and NS_{lsp}, across spaces; i.e., to represent the instances from each method in their counterpart search space. Section 4.1.3 is devoted to that comparison. Although NS_{lsp} still produces a different number of instances per solver (490 instances for Default, 140 for MPW, 2080 for MaP, and 1850 MiW), the ratio of unique instances is similar to the results from NS_{lsf} at around 30 per cent of the total amount (see Table 4.3 for more details) after 30 repetitions of the method for each solver. In fact, NS_{lsp} struggles even more than NS_{lsf} to generate instances for MPW, with only 140 instances after 30 repetitions. These results could provide more insights into the limitations discussed above; i.e., the low number of instances

for MPW could be related to the solver idiosyncrasy, and more generations of NS_{ls} or even lower thresholds restriction may improve the results.

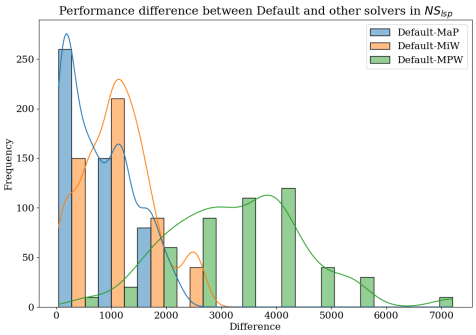
After that, the performance-gap between solvers is addressed. Designed to generate instances that are biased to certain solvers, yet diverse in the performance space, NS_{lsp} is able to generate instances with diverse performance gaps. The results are shown in Figure 4.4. Note that the distributions exhibit similar behaviour to NS_{lsf} in Figure 4.2; that is, every heuristic shows multi-modal behaviour and, despite a few instances yielding an almost negligible performance gap (close to zero), a large number of other instances obtain much higher scores (> 6000) for the MaP and MiW solvers. Even though the performance-gaps in Figure 4.4 and Figure 4.2 are considerably similar, NS_{lsp} present the advantage of not defining and calculating a set of problem-dependent features for every instance in each generation of the NS_{ls} execution. NS_{lsp} only considers diversity in the performance space and does not require any domain analysis to extract and define a set of meaningful instance features. It therefore accelerates the instance generation process.

4.1.3 Distribution of NS_{ls} in Foreign Spaces

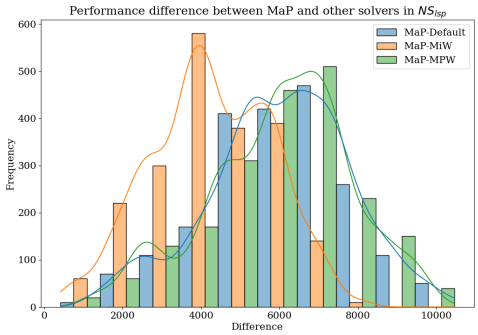
Although choosing either a feature-based or performance-based descriptor significantly impacts the design of the NS_{ls}, the results prove that NS_{lsf} is able to generate a set of diverse instances in the performance space, even though it is not designed with this intention in mind.

As a consequence, it is completely reasonable to wonder to what extent the opposite behaviour occurs; i.e., is NS_{lsp} able to generate instances that are not only diverse in the performance space, but in the feature space as well? Additionally, how do these NS_{ls} approaches compare against a pure EA algorithm, such as [108], which only considers the performance score ps (see Equation 3.2) to guide the search?

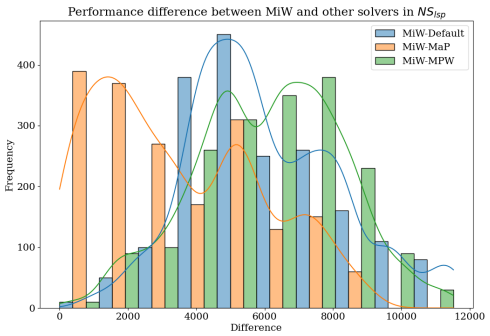
In order to address these questions, the opposite descriptor is calculated for each instance, i.e., the performance descriptor is computed for each instance generated by NS_{lsf}, and the feature descriptor is calculated for each instance generated by NS_{lsp}. Thus, for every instance the information of both descriptors is available. Besides, a base EA [108] is run 30 times to generate instances biased to the performance of each heuristic in the portfolio. To provide a convenient representation of the instances, previously created PCA models were applied to reduce the dimensionality and represent the instances in both spaces. Furthermore, it is important to note that, since the instances are evaluated in two different spaces, two PCAs were required, one per space: feature and performance. The PCA models used in this evaluation were trained with



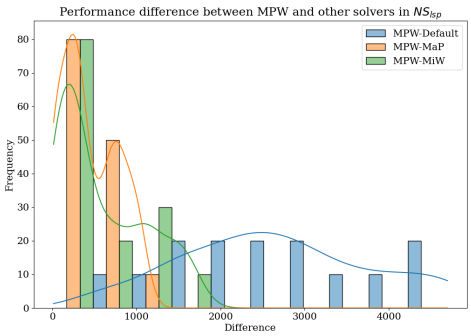
(a) Default



(b) MaP



(c) MiW

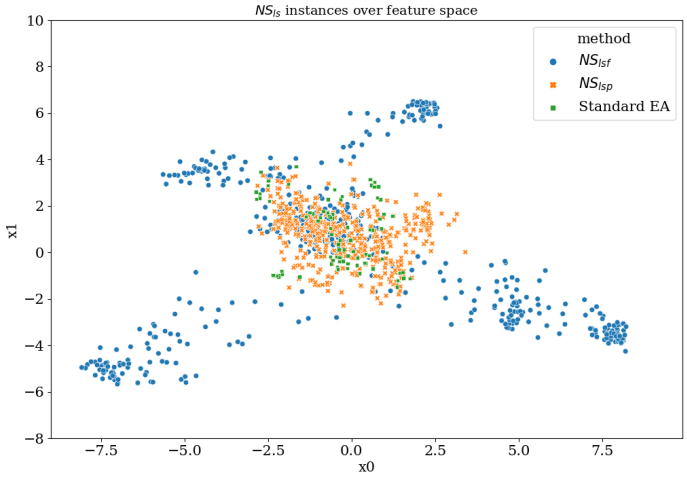


(d) MPW

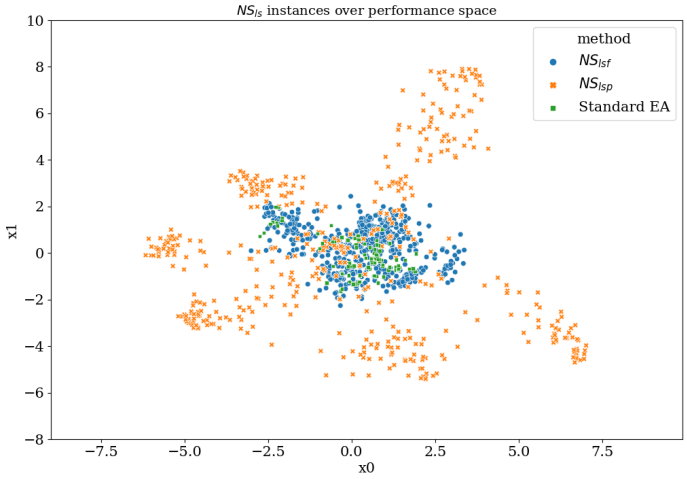
Figure 4.4 Distribution of the performance gap between the Default, MaP, MiW, and MPW approaches and other solvers in the portfolio by considering the instances generated for the former when running NS_{lsp}. The x-axis scale varies from one subfigure to another to produce better visualisations of the bars in the plots. Therefore, the differences between algorithms depend on the solver that is taken as a reference.

the data from Sections 4.1.1 and 4.1.2. Hence, the distribution of instances for each NS_{ls} variation is evaluated in both their own and foreign spaces.

Figure 4.5 provides the representation of the instances in both spaces after applying the different PCAs. Top figure represents the distribution of instances over the feature space, while the bottom figure shows the distribution of the same instances over the performance space. The colour and symbol codes used hereafter are blue dots for NS_{lsf} instances, orange crosses for NS_{lsp}, and green squares for the standard EA. Notice how the instances cover their own space more uniformly, while, in contrast, they are grouped in the centre of their foreign space. This is not surprising since NS_{ls} is designed to use each descriptor to find diversity in its own space. However, this behaviour brings even more to light regarding the importance of the descriptors in the execution of NS_{ls}. Even though the results from Section 4.1.1 indicate that



(a) Feature Space



(b) Performance Space

Figure 4.5 Instance representation in a 2D space after applying PCAs to all instances generated by both NS_{ls} approaches. The colours reflect the algorithm for instance generation.

NS_{lsf} is able to generate diverse instances in terms of performance gap to each other, Figure 4.5b shows that, as expected, NS_{lsp} still outperforms NS_{lsf} in terms of diversity in the performance space. Therefore, the descriptor must be selected conscientiously based on the kind of diversity we want to achieve. Moreover, since the standard EA algorithm tries to maximise the performance score *ps* (see Equation 3.2), the instances generated by this algorithm are located in the centre of both spaces (green squares).

Table 4.2 Space coverage using the U metric over the feature and performance spaces for instances generated by NS_{ls} and a standard EA

Method	**Feature Space**	**Performance Space**
NS_{lsf}	**0.5883**	0.4839
NS_{lsp}	0.49885	**0.6426**
Standard EA	0.4234	0.3951

The expected behaviour of the standard EA is to converge the population of instances to those regions that maximise *ps* regardless of their diversity. As a result, the set of instances provided by the standard EA is very likely to contain many redundant instances. This reasoning may explain the poor distribution of instances in the two spaces.

In order to quantitatively evaluate the distribution of instances in both spaces, the U metric (see Equation 4.2 is calculated for every method and space. Table 4.2 shows the results. First, if considering U scores as a percentage of the space covered, note how Standard EA poorly covers any space, not reaching even 0.50 of the total space. These U values support the distributions of the standard EA instances in Figure 4.5.

Next, the U values for NS_{lsf} and NS_{lsp} in their respective search spaces corroborate the distributions in Figure 4.5. NS_{lsf} is able to reach almost 0.6 similarities with a UD in the feature space while NS_{lsp} reaches 0.6426 of performance space coverage. Besides, for both methods, the U metric in the corresponding foreign space decreases drastically, not even achieving 0.50 of coverage.

Table 4.3 details the number of instances generated per solver by NS_{lsf}, NS_{lsp} and a standard EA [108] in the KP domain. In addition, the ratio of unique instances is calculated based on non-duplicated 8D feature vectors ($Unique_s$) and non-duplicated 4D performance vectors ($Unique_p$). Hence, even though NS_{lsf} and NS_{lsp} use different search spaces, both methods can be compared in terms of unique instances in a fair and more realistic way. The most interesting finding is that both methods score similar values for each unique metric in both spaces. This is a completely unexpected outcome that could probably lead to future lines of research in this work. Another important finding is that the standard EA approach [108] is able to produce sets with fewer duplicate instances (in terms of feature and performance descriptors) than NS_{ls}. However, quantitatively (Table 4.2) and qualitatively (Figure 4.5), the results indicate that the method is not able to cover as much space as NS_{ls}.

Table 4.3 Summary of the instances generated per solver and the ratio of unique instances for both NS_{lsf} and NS_{lsp} in the KP domain after 30 repetitions. Combined refers to all instances generated by a method across all targets. Unique instances are calculated based on non-duplicated 8D feature vectors ($Unique_s$) and non-duplicated 4D performance vectors ($Unique_p$).

Method	Target	Total	$Unique_s$	$Unique_p$
NS_{lsf}	**Default**	720	0.31	0.36
	MaP	2260	0.28	0.38
	MiW	2090	0.31	0.38
	MPW	220	0.40	0.40
	Combined	5290	0.28	0.38
NS_{lsp}	**Default**	490	0.36	0.38
	MaP	2080	0.30	0.39
	MiW	1850	0.31	0.38
	MPW	140	0.56	0.40
	Combined	4560	0.29	0.37
standard EA	**Default**	200	0.42	0.40
	MaP	300	0.42	0.39
	MiW	280	0.42	0.40
	MPW	110	0.50	0.40
	Combined	890	0.41	0.40

4.2 NS_{mo} Experiments

NS_{mo} is the multi-objective approach to generate diverse yet biased instances in optimisation domains. The algorithm is also run to generate KP instances as well using both previously mentioned descriptors: feature based (NS_{mof}) and performance based (NS_{mop}). In order to provide a proper comparison between NS_{ls} and NS_{mo}, most parameters remain identical to the NS_{ls} configuration provided in Table 4.4. The only exception in this parameter configuration is the elimination of ϕ, which governs the importance of ps and s in NS_{ls} (see Equation 3.3). It is not necessary for NS_{mo} since both objectives are optimised (maximised) simultaneously (see Section 3.3).

Due to the stochastic nature of NS_{mo}, each configuration is run 30 times for each of the four solvers in the portfolio and then the instances are combined in a single dataset. Additionally, the same aforementioned standardisation and dimensionality reduction (using PCA) is performed on the resulting dataset.

Table 4.4 Parameter settings for NS_{mo}, which evolves the diverse population of discriminatory instances by means of a multiobjective approach.

Parameter	**Value**
Knapsack items (N)	50
Weight and profit upper bound	1,000
Weight and profit lower bound	1
Population size	10
Crossover rate	0.8
Mutation rate	1 / ($N \times 2$)
Generations	1,000
Portfolio	Default, MaP, MPW, MiW
Repetitions (R)	1
Distance metric	Euclidean Distance
Neighbourhood size (k)	3
Thresholds (t_a, t_{ss})	3.0

4.2.1 NS_{mo} and Feature-based Descriptor

We denote NS_{mof} as the novelty search multi-objective generator which uses a feature descriptor to search for novelty. NS_{mof} is run 30 times to generate KP instances that are diverse in the feature space and biased to the performance of a specific solver in the heuristic portfolio (see Table 4.4).

The instance distribution in the feature space after the dimensionality reduction procedure is shown in Figure 4.6. Although some instances seem to slightly overlap each other, note how NS_{mof} is also able to generate four different clusters of instances, one for each solver in the portfolio. In contrast to NS_{ls}, NS_{mo} is capable of generating larger sets of instances per run. After 30 repetitions for each solver in the portfolio, NS_{mof} generated 8,770 instances, while NS_{lsf} only generated 5,290. Nonetheless, the number of instances generated per solver seems to follow a similar pattern to the NS_{lsf} results; that is, NS_{mof} was able to generate 870 instances for Default, 280 for MPW, 3420 for MaP, and 4200 for MiW after 30 repetitions of the method for each solver. The number of instances for MPW is still considerably low when compared to other solvers; in fact, NS_{mof} was only able to generate 60 more instances for MPW and 150 for Default than NS_{lsf}. Table 4.6 details the total amount and ratio of unique instances generated for NS_{mop}.

In light of the results from NS_{mof}, we can reject the assumption that the single-objective formulation of the instance generation problem may cause a low number of instances for some solvers. However, it reaffirms how (1) the stochastic nature

of the method, (2) the idiosyncrasy of each solver (making it easier or harder to generate instances biased to their performance), (3) and the need for a specific NS threshold adjustment (t_{ss}, t_a) are crucial factors that may be addressed when creating portfolios of algorithms. For instance, new approaches could consider the inclusion of more sophisticated stopping criteria, such as the number of instances in the solution set before ending the execution, rather than a certain amount of generations to be performed by NS_{ls} or NS_{mo}.

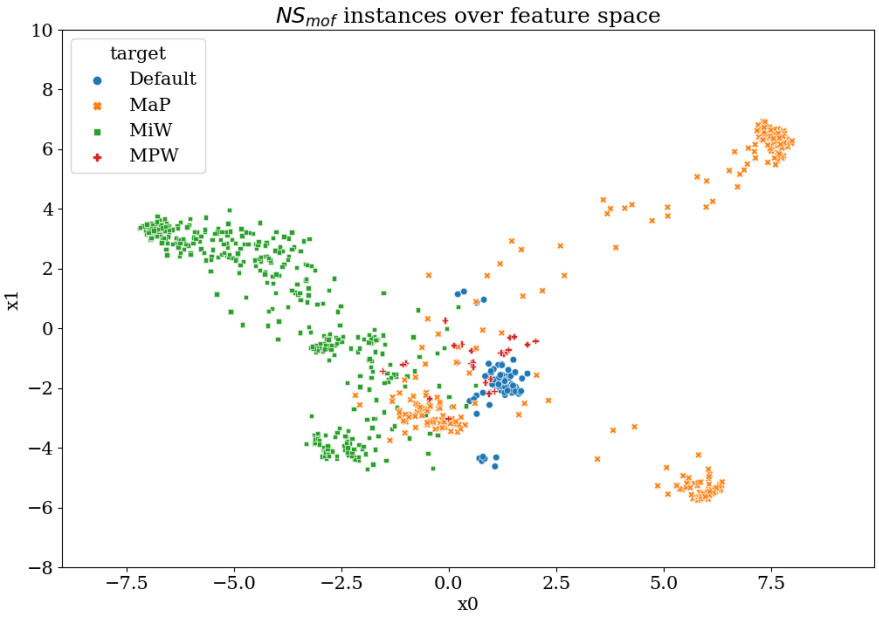
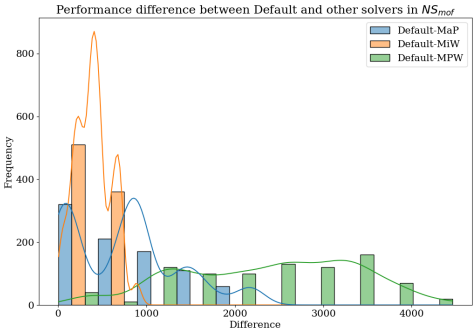


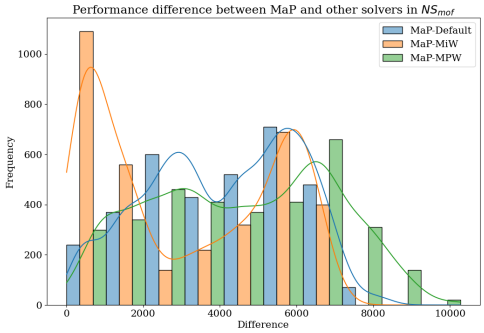
Figure 4.6 PCA is applied to a dataset containing the feature descriptors plus the instance information of all instances generated by NS_{mo} when using a feature-based descriptor to search for novelty. Blue points are the instances generated for Default, orange crosses for MaP, green squares for MiW, and red pluses for MPW.

Moreover, a quantitative evaluation of the space coverage is also provided by means of the exploration uniformity (U) metric [53, 77]. The calculation of U is detailed in Section 4.1.1.

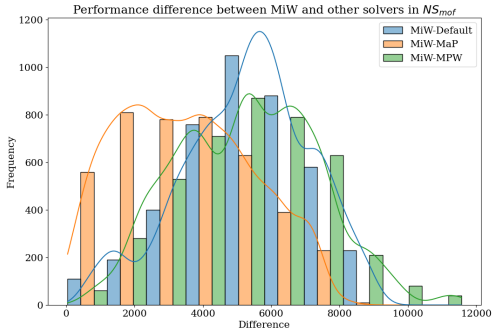
The set of instances generated by NS_{mof} scores similar space coverage to NS_{lsf}, $U = 0.5681$. Since NS_{mo} abandons the use of ϕ, it is only capable of generating a more or less spread out set of instances by tuning the thresholds (t_a, t_{ss}). Larger threshold ratios will force the algorithm to consider instances with higher s scores, and therefore a more diverse set of instances.



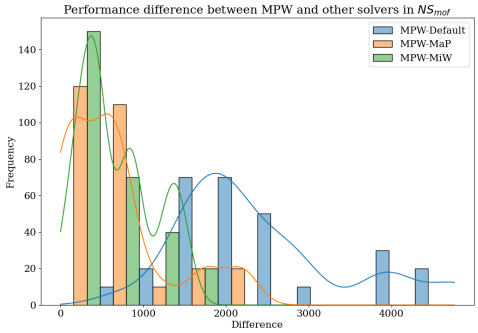
(a) Default



(b) MaP



(c) MiW



(d) MPW

Figure 4.7 Distribution of performance gap between the Default, MaP, MiW, and MPW approaches and other solvers in the portfolio by considering the instances generated for the former when running NS_{mof}. The x-axis scale varies from one sub-figure to another to produce better visualisations of the bars in the plots. Therefore, the differences between algorithms depend on the solver that is taken as a reference.

Then, a quantitative analysis of the performance-gap is also addressed for NS_{mof}. Figure 4.7 shows the distributions of the performance gap between instances for each solver. The x-axis represents the magnitude of the difference between the target solver and the other (performance gap) and the y-axis is the frequency with which the value appears; i.e., the number of instances that exhibit said performance gap. The results present similar patterns to the NS_{lsf} results in Figure 4.2. Even though NS_{mof} is not developed to generate diversity in terms of this metric, the results prove that while a significant number of instances have a relatively small gap between the performance of their target and other solvers, it is possible to find instances that exhibit performance diversity as well. The distributions for every heuristic exhibit similar behaviour to the performance diversity of NS_{lsf}. Even though a few instances have a performance gap

that is almost negligible (close to zero), a large number of other instances obtain much higher scores (> 6000) for MaP and MiW solvers.

4.2.2 NS_{mo} and Performance-based Descriptor

In a similar manner, NS_{mop} is defined as the multi-objective novelty search generator using a performance-based descriptor to generate diverse, yet discriminatory, instances. Moreover, NS_{mop} was run 30 times to generate KP instances that are biased to the performance of a specific solver in the heuristic portfolio. The same parameter settings as in Section 4.2.1 were used (see Table 4.4). Figure 4.8 shows the instance distribution over the performance space. The instances underwent the same procedure applied in previous sections. To reduce the information to two principal components, standardisation plus dimensionality reduction by means of PCA were applied to the performance descriptor and the instance information. It is rather intriguing how NS_{mo}, and in this particular case NS_{mop}, is able to find instances in several locations of the space for specific solvers, while NS_{ls} was able to generate instances in certain regions only; i.e., green squares representing instances for MiW in Figure 4.8 are located in three differentiated clusters. A multi-objective formulation of the instance generation problem may allow NS_{mo} to discover relationships and instances that NS_{ls} cannot. Apart from that, Figure 4.8 still accentuates the low number of MPW instances generated. The set of instances generated after 30 repetitions per solver of NS_{mop} contains 10,460 instances. NS_{lsp} was only able to generate 4,560 instances. Those 10,460 instances are divided into 670 instances for Default, 170 for MPW, 2220 for MaP, and 7400 MiW. Table 4.6 provides more detail about the number of instances produced for NS_{mop}.

In order to evaluate the space coverage of NS_{mop}, the U metric [53, 77] is also calculated. Thus, in the current scenario, δ is defined as the two principal components extracted from each instance after applying the aforementioned PCA model. NS_{mop} scores $U = 0.5437$, rather lower values than obtained by NS_{lsp} with $U = 0.6426$. These results may indicate that even though NS_{mop} is able to generate more than twice as many instances as NS_{lsp} in the same number of executions, the NS_{lsp} configuration used in Section 4.1.2 is able to generate a set of instances that is more spread out in the performance space.

The primary goal of using a performance-based descriptor is to generate instances that are diverse with respect to the performance space. In order to address this, the performance-gap of the instances generated by NS_{mop} is presented. Figure 4.9 provides the distributions of performance gap between instances for each solver. As before, the

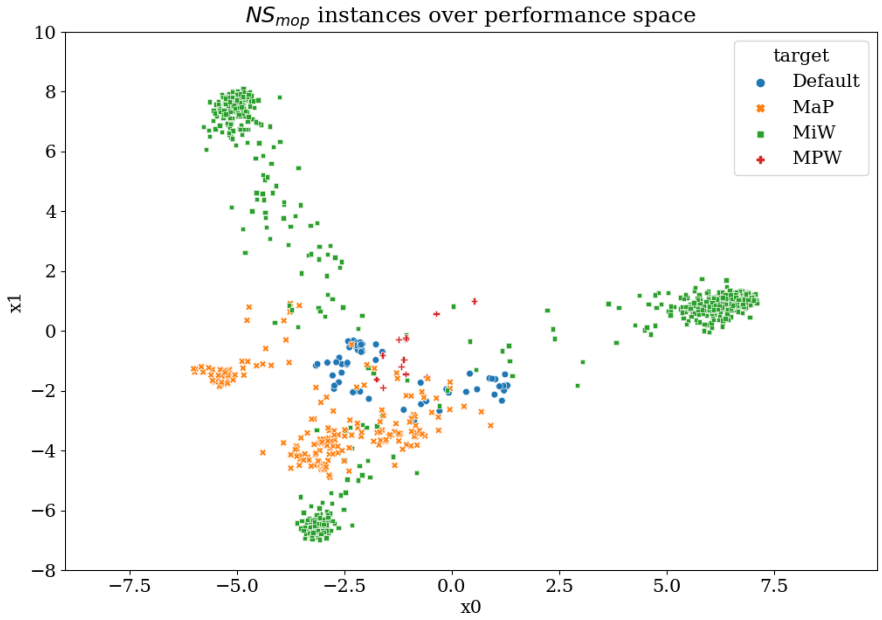
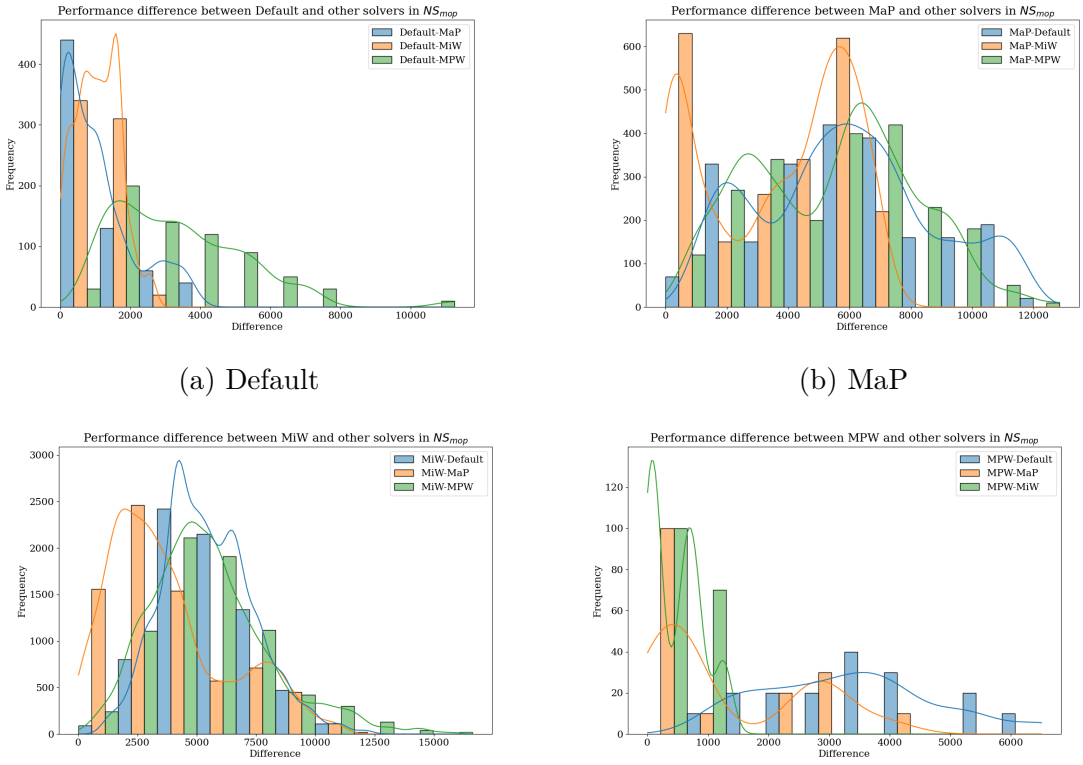


Figure 4.8 PCA is applied to a dataset containing the feature descriptors plus the information on all the instances generated by NS_{mo} when using a performance-based descriptor to search for novelty. Blue points are the instances generated for Default, orange crosses for MaP, green squares for MiW, and red pluses MPW.

x-axis represents the magnitude of the difference between the target solver and the other (performance-gap), and the y-axis is the frequency at which the value appears; i.e, the number of instances that exhibit such a performance gap. Once again, the method is able to generate diversity with respect to the performance gap between solvers. The distributions of values seem to follow a multi-modal distribution for each solver, although it can be seen more clearly for MaP (Figure 4.9b) and MiW (Figure 4.9c). In fact, NS_{mop} is able to reach higher performance gap values, almost 17,500 for MiW, than NS_{lsp}, whose highest performance gap was lower than 12,000. In addition to the higher number of instances generated per experiment, the considerably large performance gap could be a point in favour of NS_{mop} versus NS_{lsp}.



(a) Default (b) MaP

(c) MiW (d) MPW

Figure 4.9 Distribution of performance gap between the Default, MaP, MiW, and MPW approaches and other solvers in the portfolio by considering the instances generated for the former when running NS_{mop}. The X-axis scale varies from one sub-figure to another to produce better visualisations of the bars in the plots. Therefore, the differences between algorithms depend on the solver that is taken as a reference.

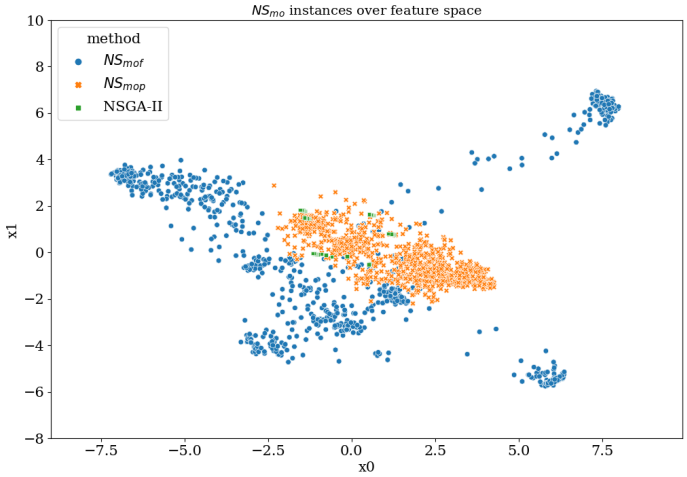
4.2.3 NS_{mo} Distribution in Foreign Spaces

Similar to the analysis conducted in Section 4.1.3 on NS_{ls} approaches, this section is devoted to evaluating NS_{mo} approaches in their respective foreign spaces; i.e., how well the instances from NS_{mof} are distributed in the performance space and vice-versa. The main question to answer here is: Can NS_{mo} generate diverse instances in a space that is not expected to, such as NS_{ls}? Moreover, both methods are compared to a multi-objective EA which solves the instance generation problem as a multi-objective optimisation problem with ps and s as the objectives to maximise. The algorithm selected for this comparison is the well-known evolutionary algorithm NSGA-II [29]. NS_{mo} is strongly based on the NSGA-II algorithm plus the addition of novelty search components and procedures such as the archive of diverse solutions and solution set.

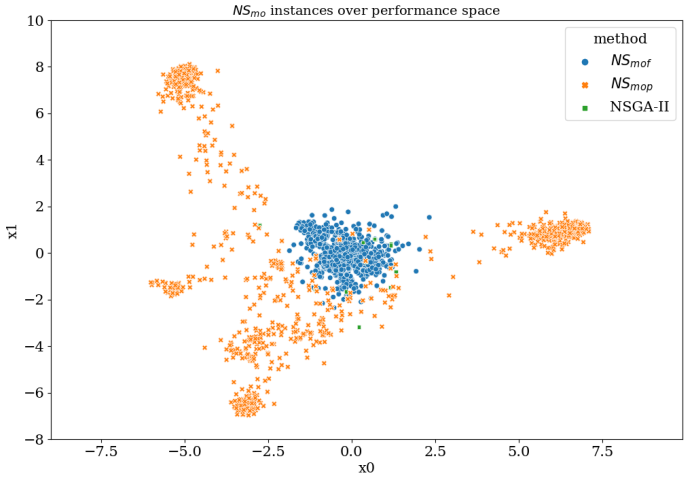
First, a dataset is created to contain all the instances generated by NS_{mof}, NS_{mop}, and NSGA-II. Then, for each instance, the opposite descriptor is calculated; i.e., a performance descriptor for the instances generated by NS_{mof}, and a feature descriptor for those generated by NS_{mop}. Therefore, every instance in the dataset is defined by both descriptors. For instances generated with NSGA-II, both descriptors are calculated.

Next, each instance was reduced to two principal components by means of PCA. Since the instances are evaluated in two spaces (feature and performance), a different PCA model has to be applied for each space. Therefore, two PCA models are required for this evaluation. Similar to the analysis performed in Section 4.1.3, we reuse the PCA models trained in Section 4.2.1 and Section 4.2.2 to transform the instances into their foreign spaces. For instance, the PCA model trained in Section 4.2.1 using NS_{mof} instances, is now used to transform instances generated by NS_{mop} by means of the recently calculated feature descriptor, and vice-versa.

Figure 4.10 presents the instances in both spaces after applying the different PCAs. The top figure represents the distribution of instances over the feature space, while the bottom figure shows the distribution of the same instances over the performance space. The colour and symbol codes used hereafter are blue dots for NS_{mof} instances, orange crosses for NS_{mof}, and green squares for NSGA-II. The results exhibit similar behaviour to NS_{ls} when compared in foreign spaces; i.e., as expected, NS_{mof} outperforms their counterpart in terms of instance distribution in their own feature space, while NS_{mop} does the same in the performance space. In addition, 560 instances generated by NSGA-II are almost imperceptibly located in the centre of both spaces. As a result of considering the instance generation problem as a pure multi-objective optimisation problem, the convergence of the algorithm results in the fact that the instances in



(a) Feature Space



(b) Performance Space

Figure 4.10 Instance representation in a 2D space after applying PCAs to all instances generated by both NS_{mo} approaches. Colours reflect the algorithm for instance generation.

the dataset are almost identical to one another. These results illustrate that even though NS_{mo} is able to generate diversity in both spaces (as discussed before in terms of distribution and performance-gap), deciding which descriptor to use is crucial and must be done conscientiously based on the expected characteristics of the instances.

Finally, it could be interesting to quantify the benefits of each method in terms of space coverage. Table 4.5 summarises the U scores for each NS_{mo} and NSGA-II

Table 4.5 Space coverage using the U metric over the feature and performance spaces for instances generated by NS_{mo} and NSGA-II

Method	Feature Space	Performance Space
NS_{mof}	**0.5681**	0.4206
NS_{mop}	0.4541	**0.5437**
NSGA-II	0.3404	0.3465

algorithm over each space. The results support the conclusions drawn from Figure 4.10b. NS_{mo} exhibit a similar behaviour as NS_{ls} in Section 4.1.3. Interestingly, NS_{ls} obtains higher U scores in every space when compared to its analogue NS_{mo} version. In addition, it seems that addressing the KP instance generation problem from a single-objective formulation benefits the space coverage. In fact, the comparison of pure evolutionary algorithms demonstrates that a standard EA ($U = 0.4234$ in the feature space, and $U = 0.3951$ in the performance space) outperforms NSGA-II ($U = 0.3404$ in the feature space, and $U = 0.3465$ in the performance space) in both spaces.

Table 4.6 Summary of the instances generated per solver and the ratio of unique instances for NS_{mof}, NS_{mop} and NSGA-II in the KP domain after 30 repetitions. Combined refers to all instances generated by a method across all targets. Unique instances are calculated based on non-duplicated 8D feature vectors ($Unique_s$) and non-duplicated 4D performance vectors ($Unique_p$).

Method	Target	Total	$Unique_s$	$Unique_p$
NS_{mof}	**Default**	870	0.24	0.17
	MaP	3420	0.23	0.31
	MiW	4200	0.26	0.35
	MPW	280	0.35	0.29
	Combined	8770	0.24	0.30
NS_{mop}	**Default**	670	0.30	0.35
	MaP	2220	0.26	0.34
	MiW	7400	0.24	0.30
	MPW	170	0.35	0.36
	Combined	10460	0.23	0.30
NSGA-II	**Default**	100	0.19	0.14
	MaP	210	0.2	0.14
	MiW	210	0.2	0.14
	MPW	40	0.63	0.4
	Combined	560	0.22	0.16

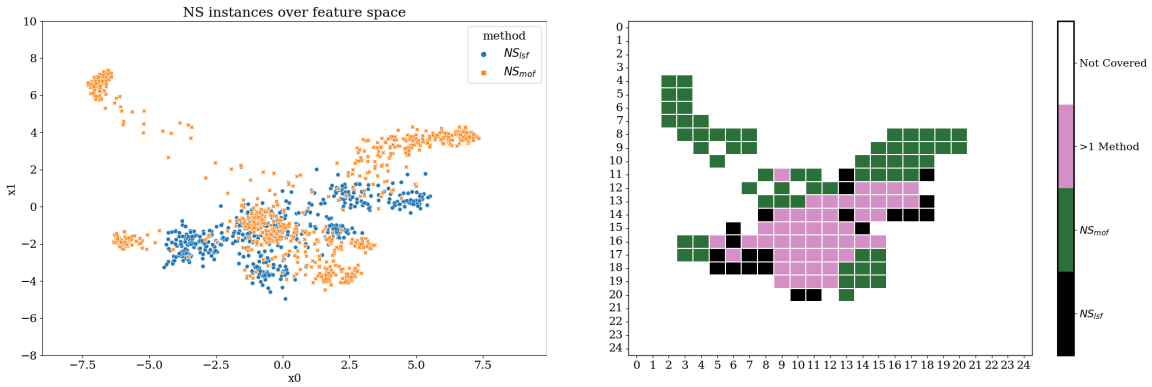
Table 4.6 details the number of instances generated per solver by both NS_{mo} variants in the KP domain. The ratio of unique instances is calculated based on non-duplicated 8D feature vectors ($Unique_s$) and non-duplicated 4D performance vectors ($Unique_p$). It seems that the larger sets of instances generated by NS_{mo} variants come at the cost of more redundant instances (when compared by descriptors). NSGA-II results, apart from the MPW discriminatory instances, are considerably lower than any NS_{mo} variant not only in terms of unique ratios but also the total amount of instances.

4.3 Comparison between NS_{ls} and NS_{mo}

The previous sections were devoted to evaluating each of the methods proposed in Chapter 3 with different instance descriptors. Now, NS_{ls} and NS_{mo} are compared to each other to evaluate whether one method gives better results in terms of instance distribution, or if it might be beneficial to construct an ensemble of the two approaches. In order to evaluate both methods, two datasets are created, one named *features dataset*, with instances generated by NS_{lsf} and NS_{mof}, and the other named *performance dataset*, with instances from NS_{lsp} and NS_{mop} executions. Afterwards, each dataset was processed using the standardisation and dimensionality reduction procedure described in the preceding sections. Similar to Sections 4.1.3 and Section 4.2.3, different PCA models were applied to each dataset based on the instance descriptor. For instance, the features dataset was processed using a PCA model to reduce the feature-based descriptor and the instance information from each instance to two principal components. An analogous process was used for the performance dataset with a PCA for the performance-based descriptor plus the instance information. Figure 4.11 shows the distribution of instances from each method over the feature space. While the left side provides the traditional representation used in this chapter, the right side shows the representation of instances in terms of filled cells of the space. The space is divided into a grid of 25×25 cells, as done when calculating the exploration uniformity (U) metric. Next, the number of cells filled by each method, NS_{lsf} and NS_{mof}, is calculated. For example, a cell is considered filled by a method if at least one instance is located inside the cell. The right side of Figure 4.11 provides a graphical representation of this calculation in the feature space.

Black designates the cells filled only with instances generated by NS_{lsf}, green cells are those filled only by NS_{mof}, and pink represents the cells that are filled by instances from both algorithms. The white cells are those which are not filled by any method. Furthermore, in terms of filled cells, NS_{lsf} was able to fill 69 cells, NS_{mof} 109, and an ensemble of both methods filled 130 cells. It seems that an ensemble approach could be slightly beneficial in terms of space coverage. In fact, the filled-cells calculation procedure reveals that there is only an overlap of 48 cells between NS_{ls} and NS_{mo}.

Alternatively, Figure 4.12 provides similar results for the performance dataset. Following the previous procedure, the performance space was also divided into a grid of 25×25 cells and a similar filled-cells counting procedure was applied. The results indicate that NS_{lsp} filled 60 cells in a 25x25 performance space, NS_{mop} 98 cells, and the ensemble of NS_{lsp} and NS_{mop} methods 118 cells. Moreover, the overlap of filled cells between NS_{lsp} and NS_{mop} is quite similar to before, with 40 cells.

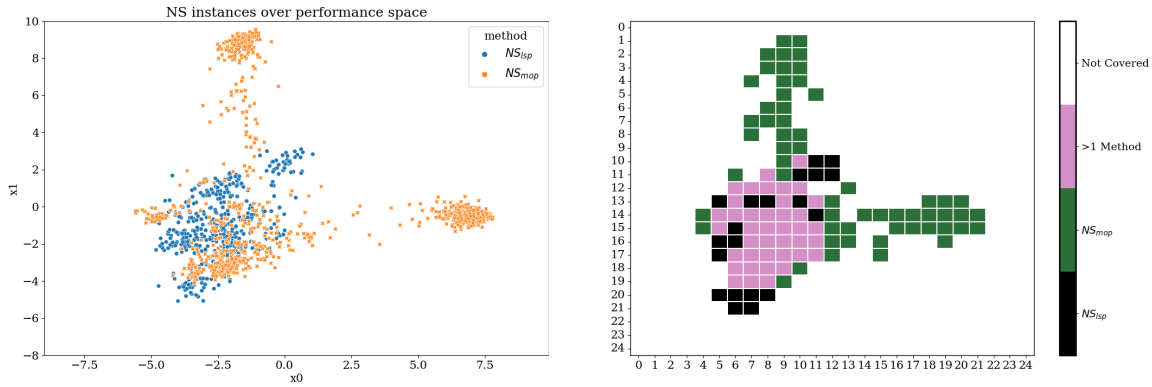


(a) Feature space distribution

(b) Feature space overlap

Figure 4.11 Instances generated for different NS algorithms represented in the same feature space. The left-hand side represents the distribution of instances where blue dots represent the instances from NS_{lsf} and orange squares for NS_{mof}. The right-hand side represents cells filled by each method in the 25 $\times$ 25 grid obtained from all the information available. Black cells are only filled with instances from NS_{lsf}, green cells are only filled by NS_{mof}, and pink represents those cells that are filled by instances from both algorithms. White cells are those which are not filled by any method.

In conclusion, it seems that constructing an ensemble of both algorithms could reinforce instance generation. On the one hand, NS_{ls} provides slightly better results in terms of U scores and instance distribution across spaces. On the other hand, NS_{mo} is able to generate larger sets of instances per execution with U scores close to NS_{ls}. For that reason, and considering the low overlap exhibited in Figures 4.11 and 4.12, balancing the generation of instances for the KP domain between the linear-weighted NS and the multi-objective NS may improve the results in terms of space coverage, instance distribution and amount of data.



(a) Performance space distribution (b) Performance space overlap

Figure 4.12 Instances generated for different NS algorithms represented in the same performance space. The left side represents the distribution of instances, where blue dots represent the instances from NS_{lsp} and orange squares for NS_{mop}. The right side represents cells filled by each method in the 25 × 25 grid obtained from all the information available. Black cells are only filled with instances from NS_{lsp}, green cells are only filled by NS_{mop}, and pink represents the cells that are filled by instances from both algorithms. White cells are those which are not filled by any method.

4.4 Summary

The main objective of this chapter is to assess the extent to which Novelty Search can be used to generate diverse but discriminatory instances for the KP with respect to a portfolio of solvers. In particular, two major variants of an NS algorithm were considered: NS_{ls}, which uses a weighted combination of performance and diversity to drive the evolution of instances, and NS_{mo}, which solves the problem by optimising performance and diversity simultaneously to drive the evolution of instances. At the same time, each variant can be used with two instance descriptors: feature-based or performance-based. The results revealed that both NS methods were able to provide competitive results in terms of coverage of both spaces. However, as expected, each descriptor was able to perform better in its own space. Besides, a portfolio of four deterministic heuristics specially designed for the KP domain was considered for generating instances. However, previous work has demonstrated the application of an EA-based portfolio for the same domain [90]. One of the main advantages of the methods proposed is that a *set* of diverse and discriminatory instances is returned in a single run. In contrast, existing methods in the literature, as previously stated [2, 108, 122], need to be run repeatedly to generate multiple instances, given that the EAs often converge to a single solution;

furthermore, there is no guarantee that repeated runs will deliver unique solutions. To evaluate that, the results from the NS methods were compared with standard EA approaches, like those proposed by Plata *et. al* [108] for NS_{ls} and NSGA-II for the multi-objective variant NS_{mo}.

Finally, several experiments were conducted with both NS approaches not to only provide a quantitative and qualitative evaluation of the diversity of the instances generated with respect to the feature space of the KP instances and the performance space of the solvers for the KP, but also to evaluate the potential of constructing an ensemble of both methods to boost the quality of the results.

Chapter 5

Generating instances for the TSP domain

5.1 Background

The Travelling Salesman Problem (TSP) is arguably the most well-known NP-hard combinatorial optimisation problem [40]. The TSP has been studied in depth in academic and real-world scenarios due to the multiple applications and variants that can be derived from the core formulation [20, 71, 84]. Some real-world applications of TSP variants are computer and printed circuit board (PCB) wiring [133], vehicle routing (such as Garbage Collection or Unmanned Aerial Vehicles) [71, 103], clustering a data array or job-shop scheduling without intermediate storage [7, 80]. In the present thesis, the TSP domain evaluated is the Symmetrical Travelling Salesman Problem.

Definition 3 *Given a set of N nodes or vertices G and their pairwise distances D to each other, such as $D(i,j) = D(j,i)$ and $D(i,i) = 0$, the goal is to find the shortest tour T that starts and ends at a pre-defined node and visits every other node exactly once.*

D is a distance matrix that stores the distances between each pair of nodes $(i,j) \in G$. Figure 5.1 shows a real-world example of a TSP instance, where the goal is to travel across Italy visiting each location once. The right side provides the shortest tour T that solves the problem.

Switching from one domain to another requires redefining several concepts. For instance, how the instances are represented in memory, which metric is used to calculate the performance score *ps*, or what descriptor the NS should consider to compute the

Figure 5.1 (a) An example of a symmetrical TSP instance with $N = 20$ nodes on the map of Italy. The right side (b) provides the optimal solution to the instance. The graphics are obtained from [56].

sparseness s of a given instance during the evolutionary process. The literature on feature extraction has proposed several descriptors for a TSP instance [9, 122]. However, the experimental evaluation for the TSP is primarily focused on demonstrating the domain-switching capabilities of the methods by means of performance-based descriptors. Therefore, the calculation of s for a TSP instance is similar to the previous examples for the KP domain when running NS_{lsp} or NS_{mop} (see Sections 4.1.2 and 4.2.2 respectively).

Moreover, although the TSP is a sequence or permutation-based problem, an instance of the TSP can be stored in memory similarly to a KP instance. Figure 5.2 shows the representation of a TSP instance in memory. Equivalent to Figure 3.3, the pairs (w_i, p_i) have been replaced for pairs of (x_i, y_i) coordinates where each pair i represents the spatial location of the ith-node of the instance.

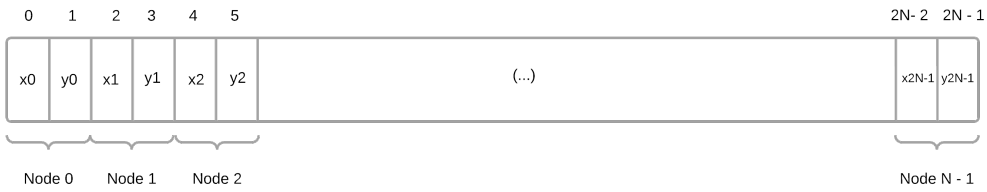


Figure 5.2 TSP Instance Representation as Stored in Memory (Genotype).

Computing the performance score ps also requires some adjustments for the new domain. In the KP domain, ps is calculated by means of Equation 3.2, using a metric based on the profit obtained by the solvers for a certain instance (see Section 3.1.2). On the other hand, even though we can reuse the same formulation to calculate ps in

Algorithm 6: Greedy TSP Heuristic

```
   Input : N, M
1  sort_egdes(M);
2  T = ∅; while |tour| ≠ N do
3  |   e = select_shortest_edge(M);
4  |   if e is valid then
5  |   |   T = T ∪ {e} ;
6  |   end
7  end
8  return T;
```

the TSP domain, defining a new solver-performance metric is necessary. Thus, *profit* is replaced by a new metric known as *Inverse Tour Length* (IL). The IL metric is calculated as:

$$IL = \frac{1.0}{L} \tag{5.1}$$

where L is the tour length that a solver obtains for a certain TSP instance.

As a result, *ps* for the TSP is calculated using Equation 5.2. t_{IL} is the Inverse Tour Length achieved by the target algorithm while o_{IL} is the set of IL scores obtained by the remaining solvers in the portfolio.

$$ps = t_{IL} - max(o_{IL}) \tag{5.2}$$

5.1.1 Portfolio of Solvers for the TSP Domain

An important difference with respect to the parameter settings for the KP domain lies in the portfolio of solvers. There is a rich literature of algorithms for the TSP domain which goes from exact solvers such as Dynamic Programming, to heuristics and meta-heuristics [8, 33, 102, 111, 139]. However, most of those solvers are more computationally complex than the KP heuristics in the previous portfolio, resulting in considerably larger run times when evaluating instances. Hence, to reduce the duration of the runs, the new portfolio contains three well-known TSP heuristics [8, 102]. First, we have a Greedy heuristic that constructs a tour for a TSP instance by progressively selecting the shortest edge and adding it to the tour. However, in order to include a new edge to the tour, it must not create a cycle with fewer than N edges (where N is the number of nodes to visit in the instance), visit any node more than once or insert the same edge twice. Algorithm 6 outlines the Greedy heuristic for the TSP.

Algorithm 7: 2-Opt TSP Heuristic

```
   Input : N, M
 1 T = create_initial_tour(M);
 2 while improve do
 3 |   for i = 0 to N − 1 do
 4 |   |   for j = i + 1 to N do
 5 |   |   |   T′ = 2_opt_swap(T, i, j);
 6 |   |   |   if T′ < T then
 7 |   |   |   |   T = T′;
 8 |   |   |   end
 9 |   |   end
10 |   end
11 end
12 return T;
```

Algorithm 8: 2-Opt Swap Operator

```
   Input : T, i, j
1 T′ = ∅;
2 T′[0 : i] = T[0 : i];
3 T′[i + 1 : j] = reverse(T[i + 1 : j]);
4 T′[j : N − 1] = T[j : N − 1];
5 return T′;
```

We then consider two variants of the 2-Optimal (2-Opt) heuristic [8, 23, 102]: the standard 2-Opt (2-Opt) and a Nearest-Neighbour 2-Opt (NN2-Opt). 2-Opt is a local search-based heuristic that attempts to make local improvements to pre-existing tours. For instance, from a previously created tour T, the 2-Opt heuristic removes two edges and reconnects the two paths. Note that a new connection would be considered valid if and only if it does not create two disjoint cycles with fewer than N edges and the new tour T' is shorter than T. The algorithm continues making that modification until no improvements are found.

Algorithms 7 and 8 outline the 2-Opt heuristics for the TSP and a swap operator to reconnect the new paths created.

The only difference between 2-Opt and NN2-Opt algorithms lies in the construction of the initial tour T before running the 2-Opt local search. The standard 2-Opt algorithm creates an initial tour T by randomly selecting the order of the nodes to visit. After that, 2-Opt is applied on T. Similarly, NN2-Opt, creates T by running another TSP heuristic, the *Nearest Neighbour* (NN) heuristic [8, 102]. NN is a straightforward heuristic that from the start node, creates a tour T by selecting the nearest not

visited node until $|T| = N - 1$, then it returns to the starting point. The portfolio is constructed in such a way that we can evaluate whether the method is able to differentiate both 2-Opt variants from each other and the Greedy Heuristic. Previous work in the KP domain using a portfolio of different EA configurations proves that the method is able to find distinct instances for each configuration [90], even though only one parameter is changed from one configuration to another. Hence, the question is whether a different initialisation process in a 2-Opt heuristic is enough to distinguish between the two solvers.

5.1.2 Parameter Tuning

Table 5.1 Parameter settings for NS_{ls}, which evolves the diverse population of discriminatory instances for the TSP.

Parameter	**Value**
Number of nodes (N)	50
(x, y) coordinates minimum values	1
(x, y) coordinates maximum values	100
Population size	10
Crossover rate	0.8
Mutation rate	1 / ($N \times 2$)
Generations	1,000
Portfolio	Greedy, 2-Opt, NN2-Opt
Repetitions (R)	1
Distance metric	Euclidean Distance
Neighbourhood size (k)	5
ϕ	0.0, 0.15, 0.30, 0.40, 0.50, 0.60, 0.70, 0.85, 1.00
Thresholds (t_a, t_{ss})	1e-06

In terms of parameter settings, both NS algorithms are configured similarly to those in previous sections. Table 5.1 summarises the parameter values for both methods. Note that the threshold values t_{ss} and t_a are updated to the new domain as well. New values are necessary since we are using a performance-based descriptor and a reformulated ps calculation procedure (see Equation 5.2), which results in considerably lower ps values.

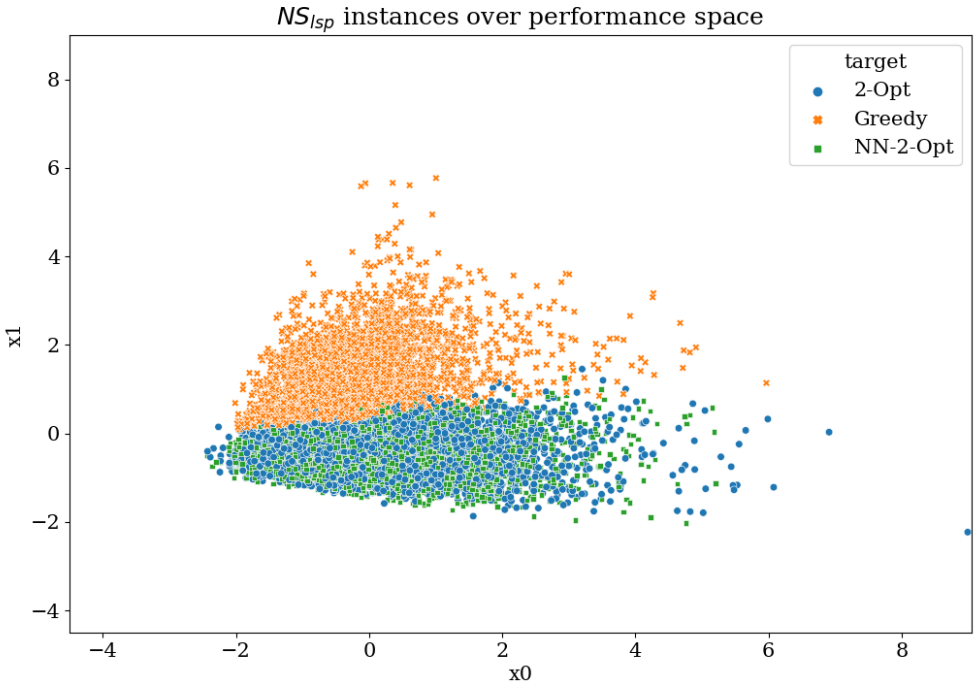


Figure 5.3 Instance representation in a 2D space after applying PCAs to all instances generated by NS_{lsp} using different ϕ values. Colours and symbols reflect the 'winning' solver for each instance. Blue dots are the instances generated for 2-Opt, orange crosses represent instances produced for Greedy, and green squares are the instances for NN-2-Opt.

5.2 Generating TSP Instances with NS_{lsp}

5.2.1 Impact of ϕ on NS_{lsp} for the TSP Domain

Following similar evaluations performed for the KP in previous work [90], let us now consider the impact of ϕ in the distribution of the instances across the performance space. A total number of nine different values are selected for ϕ from a wide range of values, such as $\phi \in \{0.0, 0.15, 0.3, 0.4, 0.5, 0.6, 0.7, 0.85, 1.0\}$. Note that $\phi = 0.0$ will eliminate the ps from the calculation of the fitness in NS_{lsp} (see Equation 3.3), and $\phi = 1.0$ does the same for s. NS_{lsp} was run 30 times for each of the three solvers and for every different ϕ value. Thus, NS_{lsp} performed 810 independent executions to generate different sets of TSP instances, and then the results were combined in a single dataset.

Figure 5.3 shows the distribution of all instances across the performance space after applying a dimensionality reduction procedure. Similarly to the KP domain, the dimensionality reduction procedure consists of standardising the information of the TSP instances and then reducing them to just two principal components by means of PCA. It is important to note that, for the TSP domain, PCAs are trained using only the performance of each solver over the instances. It is rather interesting to see how NS_{lsp} is able to separate the two main solvers in the portfolio. Although 2-Opt and NN-2-Opt could be considered the same algorithm at their core (since they actually are), the initialisation procedure gives them different flavours of the same heuristic and, in fact, they do perform differently, as we will see. Even though Figure 5.3 shows all ϕ-sets of TSP instances, Appendix B.1 includes a separate plot for each ϕ-set which illustrates the same behaviour. Referring back to the question of whether a distinct initialisation procedure would be enough to differentiate the two 2-Opt methods, the results may indicate that it is not. Even though the initialisation generates different initial tours T, it seems that it has not had enough impact on the performance diversity of the algorithm to generate instances clearly located in different regions of the performance space, i.e., both approaches provide the same performance when solving instances located in the same region. However, a more in-depth parameter tuning in NS_{lsp} could lead to better separation of the instances in diverse clusters.

Figure 5.3 shows how a cluster of instances for 2-Opt flavoured solvers is clearly differentiated from the instances generated for the Greedy algorithm. Even though there is no separation between the two, this may be a result of setting considerably low threshold values t_{ss} and t_a, which could have included instances in the solution set that have low novelty values. However, low threshold values such as 1e-06 for NS_{lsp} are necessary, since it relies on the new *ps* formulation (see Equation 5.2), which generates considerably smaller performance scores than the KP domain formulation.

Moreover, to genuinely evaluate the impact of ϕ on the distribution of instances over the space, we must evaluate the instances based on exploration uniformity (U). To do this, the dataset is divided into nine subsets, each containing instances generated with one of the ϕ values tested. Then, the U metric is calculated for every subset. Similarly to the KP domain, the space defined by the limits of the *combined dataset* is divided into a grid of 25×25 cells, after which the number of instances in each cell is counted. The rest is calculated as described before. Table 5.2 summarises the total number of instances generated after running NS_{lsp} 30 times per solver and the ϕ value, as well as the U metric for each ϕ value as well as the combined set of instances. Note how, as expected, there exists an inverse correlation between the performance

Table 5.2 Summary of the instances generated per solver, performance space coverage in terms of the U metric, the total amount of instances generated and the ratio of unique instances for NS_{lsp} when setting different values for ϕ.

ϕ	**2-Opt**	**Greedy**	**NN-2-Opt**	**Total**	$Unique_p$	**U**
0.00	187	196	200	583	0.7975	0.5900
0.15	348	308	290	946	0.8097	**0.6147**
0.30	624	536	579	1739	0.9465	0.6016
0.40	769	483	783	2035	0.9769	0.5737
0.50	694	426	588	1708	0.9900	0.5397
0.60	511	272	479	1262	**1.0000**	0.4937
0.70	427	284	499	1210	0.9958	0.4929
0.85	575	407	594	1576	0.9987	0.4738
1.00	339	265	404	1008	0.9970	0.4733
Combined	4474	3177	4416	12067	0.9558	0.5859

space coverage and ϕ. The larger ϕ becomes, the less space is covered by the resulting set of instances. In fact, Pearson's Correlation Coefficient [73] (r) between ϕ and U score is $r = -0.9234$, an almost perfect negative linear relationship between the two variables. Moreover, Figure 5.4 not only shows the relationship curve between these variables, it also illustrates the ratio of unique instances generated between the different sets of instances. Here, unique instances are calculated based on non-duplicated 3D performance descriptors ($Unique_p$). We use the ratio of unique instances rather than just the total amount of unique instances to fairly compare sets of different sizes. Also interesting is the behaviour of the unique instances in the TSP domain when evaluated against ϕ. Although one may intuitively think that higher ϕ values would tend to produce fewer unique instances per run, $r = 0.8313$ indicates that there exists a strong positive correlation between ϕ and the ratio of unique instances that NS_{lsp} produces for this portfolio in the TSP domain. This metric increases as ϕ does, reaching its maximum value when $\phi = 0.6$ with a set of 1,262 unique instances. From these data, we can extract the following conclusions for the behaviour of NS_{lsp} in this scenario: (1) as ϕ increases, NS_{lsp} is prone to produce sets of instances that tend not to uniformly cover the performance space, (2) counter-intuitively, the results suggest that higher ϕ values can produce fewer redundant sets based on the genotype of the instances (see Figure 5.2). Moreover, for this scenario, it seems that ϕ values within the range 0.3 to 0.4 may produce the best sets of instances in terms of a trade-off between U and the ratio of unique instances. With respect to the number of instances generated, although there exists a slight variation in the number of instances for different ϕ settings, there

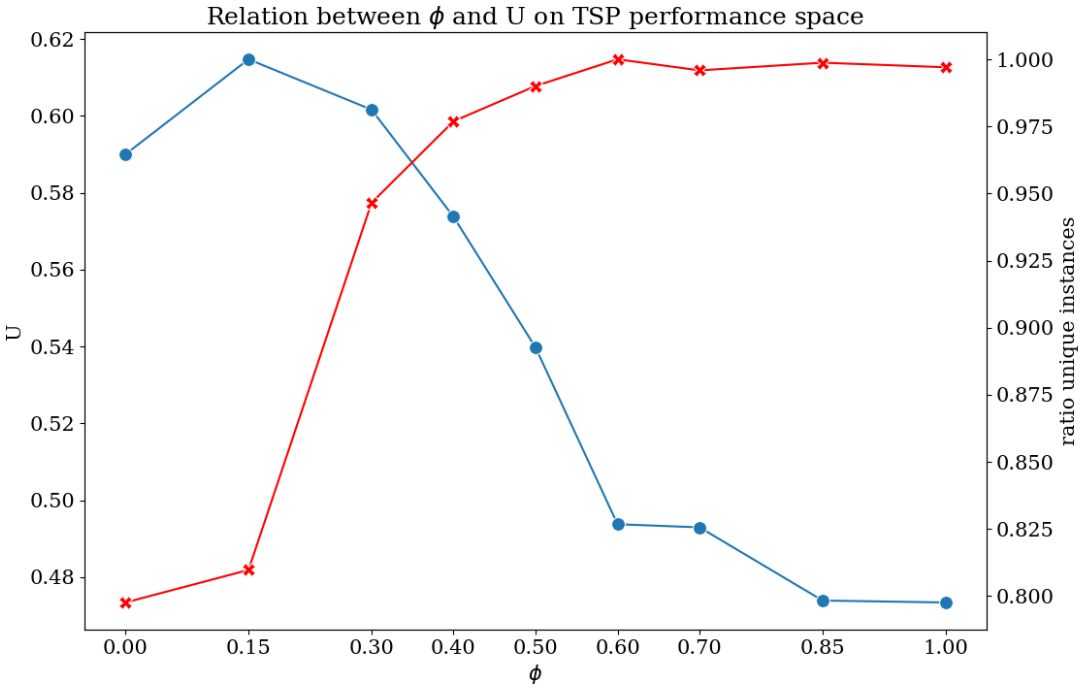
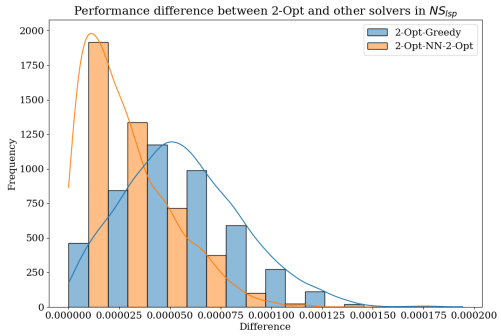


Figure 5.4 Relationship between ϕ settings for NS_{lsp} and generated TSP instances, the U scores (blue dots), and the ratio of unique instances (red crosses) generated in the resulting set of instances.

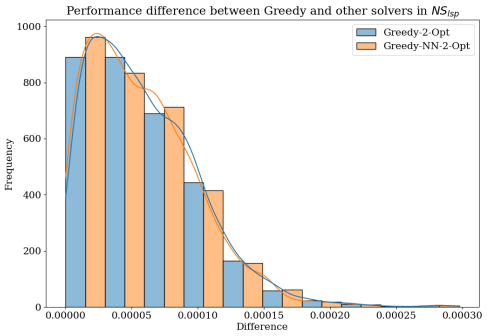
is no noticeable dissimilarity in the total instances generated for each of the three solvers in the portfolio, as is the case of MPW in the KP domain.

In terms of performance gap differences, Figure 5.5 presents the distributions between each solver in the combined dataset. Notice how there do indeed exist differences in the magnitude of the performance gap among the instances generated for the 2-Opt solver variants (see Figures 5.5a and 5.5c). Besides, 2-Opt solvers show slight differences in performance even for Greedy instances (Figure 5.5b). It is important to note that the performance gap for the TSP domain is based on the IL metric (see Equation 5.1). Thus, the difference values on the X-axis are rather lower than in the KP domain. This thus proves that, similar to the previous work in the KP domain [90], the method is able to generate not only diverse but also biased instances even for different configurations or variants of the same core solver.

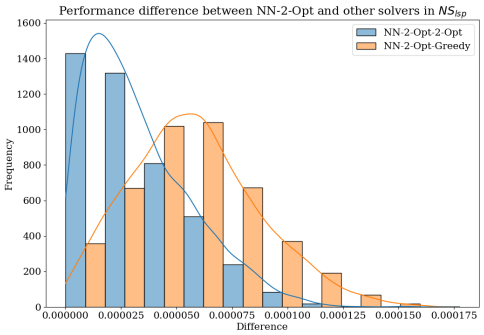
For more detail about the distribution of instances over the performance (such as Figure 5.3), as well as the performance gap histograms, for every other ϕ-subset of instances, refer to the Appendix B of this thesis.



(a) 2-Opt



(b) Greedy



(c) NN-2-Opt

Figure 5.5 Distribution of performance gap between the 2-Opt, Greedy, and NN-2-Opt approaches versus other solvers in the portfolio by considering all ϕ-generated instances for the former when running NS_{lsp} for the TSP domain. The x-axis scale varies from one sub-figure to another to better display the bars in the plots. Therefore, the differences between algorithms depend on the solver that is taken as a reference.

5.3 Generating TSP Instances with NS_{mop}

Table 5.3 Parameter settings for NS_{ls}, which evolves the diverse population of discriminatory instances for the TSP.

Parameter	**Value**
Number of cities items (N)	50
(x, y) coordinates minimum values	1
(x, y) coordinates maximum values	100
Population size	10
Crossover rate	0.8
Mutation rate	1 / ($N \times 2$)
Generations	1,000
Portfolio	Greedy, 2-Opt, NN2-Opt
Repetitions (R)	1
Distance metric	Euclidean Distance
Neighbourhood size (k)	5
Thresholds (t_a, t_{ss})	1e-06

In order to exemplify the domain transition in both methods, NS_{mop} is run 30 times per solver to generate different subsets of TSP instances. Then, as before, the instances are combined into a single dataset. The parameter setting for NS_{mop} is summarised in Table 5.3. Note that the difference with the NS_{lsp} configurations is the removal of ϕ.

Figure 5.6 shows the distribution of the instances over the performance space. In contrast to the NS_{lsp} results in Figure 5.3 or any other figure in Appendix B, the distribution of instances from NS_{mop} over the performance space exhibits behaviour that is difficult to classify properly. Apart from two distinct clusters of NN-2-Opt instances located in the lower right and upper left corners of the space, most instances are intermingled within a cluster in the lower left corner. Even though throughout this thesis we have used the same dimensionality-reduction technique (with the aim of producing the fairest possible comparison), perhaps more complex techniques may be more suitable for this scenario. We should also mention that while NS_{lsp} was only able to generate instances for 2-Opt solvers that are located in the same region of the space, NS_{mop} was able to find several drastically different instances for both solvers. In terms of the number of instances generated per method, NS_{mop} produced 487 instances for 2-Opt, 410 for Greedy, and 2,582 for NN-2-Opts after 30 runs of the method for each solver. NS_{mop} generated more than half of the instances generated by NS_{lsp} for the NN-2-Opts solver after completing the ϕ parameter tuning evaluation from Section 5.2.1, while obtaining a similar amount of instances to most $\phi - NS_{lsp}$

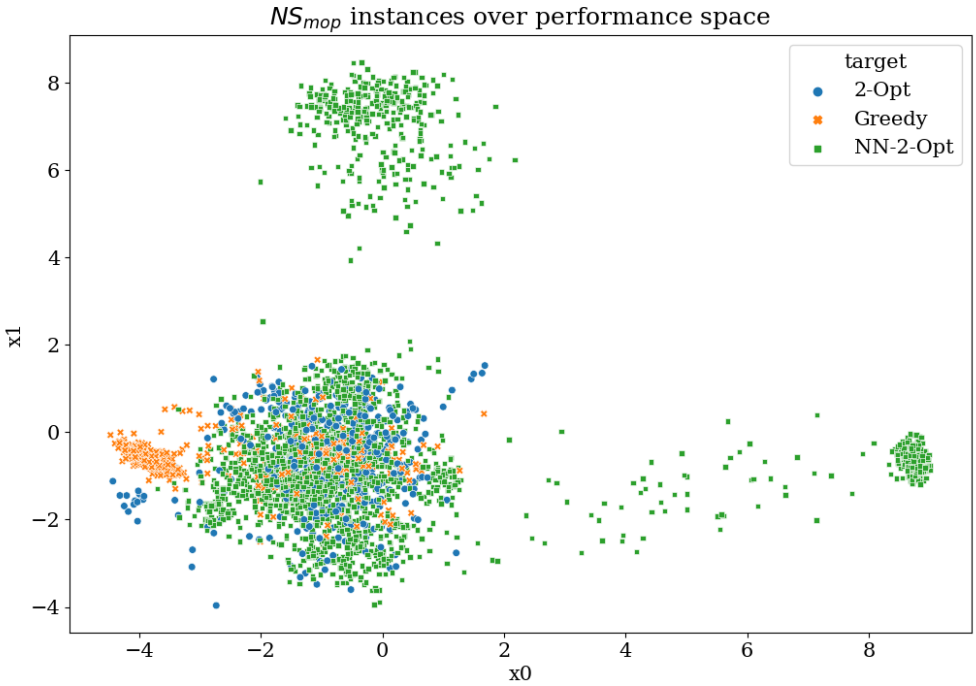
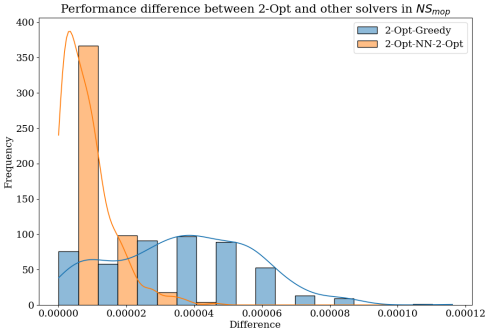


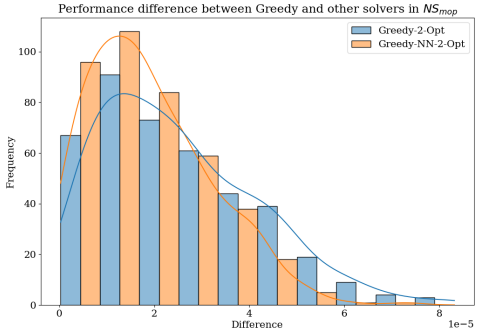
Figure 5.6 Instance representation in a 2D space after applying PCAs to all the instances generated by NS_{mop}. The colours and symbols reflect the 'winning' solver for each instance. Blue dots are the instances generated for 2-Opt, orange crosses Greedy, and green squares are the instances for NN-2-Opt.

configurations for 2-Opt and Greedy. The U metric is calculated for NS_{mop} following the same procedure described before in this thesis. NS_{mop} scores $U = 0.5702$, providing better results than five out of nine NS_{lsp} configurations from Section 5.2.1, in particular, those using ϕ values in the range 0.5 to 1. However, NS_{lsp} may still be preferable based on the better distribution over the performance space.

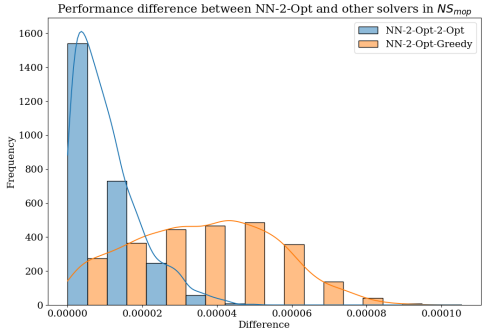
Finally, Figure 5.7 shows the performance gap distribution between solvers. Note that despite the poor instance distribution over the performance space, there exists slight diversity in terms of the performance gap between the instances generated by NS_{mop}. However, NS_{lsp} seems to obtain more diverse instances based on the performance gap. Nevertheless, there is one interesting outcome from NS_{mop} results in the TSP domain: the instances generated for NN-2-Opt. NS_{mop} obtained instances for NN-2-Opt that are really noteworthy, not only in terms of the large number of instances generated, but also the particular distribution over the performance space. While NS_{lsp} grouped the instances from 2-Opt type solvers, NS_{mop} was able to find



(a) 2-Opt



(b) Greedy



(c) NN-2-Opt

Figure 5.7 Distribution of performance gap between the 2-Opt, Greedy, and NN-2-Opt approaches versus other solvers in the portfolio by considering the instances generated for the former when running NS_{mop} for the TSP domain. The x-axis scale varies from one sub-figure to another to better display the bars in the plots. Therefore, the differences between algorithms depend on the solver that is taken as a reference.

distinctive characteristics for instances that are diverse and yet biased to the NN-2-Opt solver performance. Perhaps a more in-depth parameter tuning evaluation of NS_{mop} can provide better results in terms of instance distribution and performance diversity for other solvers as well.

Finally, Figure 5.8 illustrates the overlap between the two methods in a 25 x 25 grid over the performance space. It is also important to note that a combination of the two methods does not produce better space coverage values ($U = 0.5725$), with an unnoticeable difference from NS_{mop} ($U = 0.5702$).

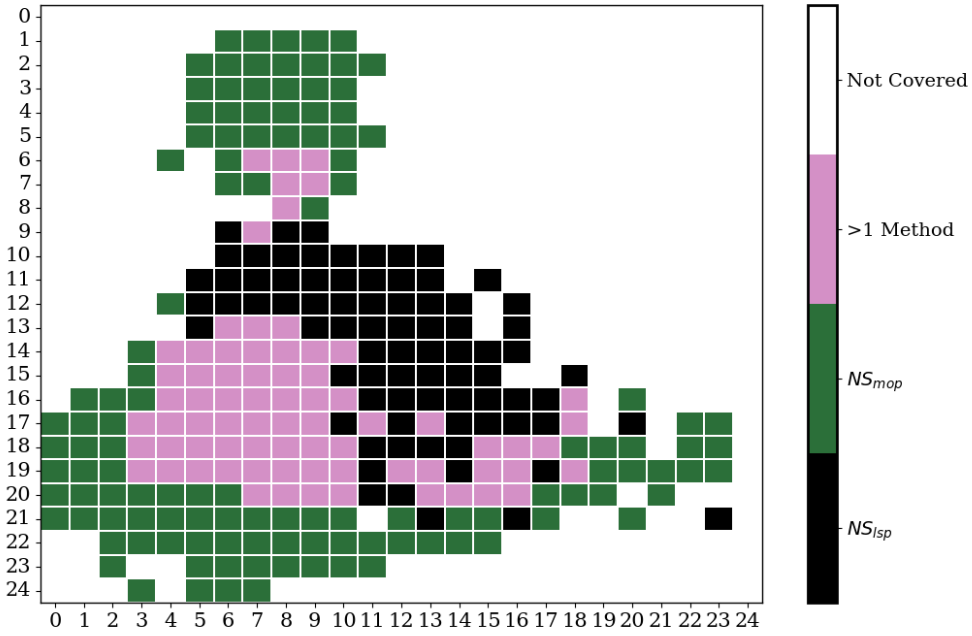


Figure 5.8 Instances generated by different NS algorithms are represented in the same performance space. The cells filled by each method in the 25 × 25 grid are obtained from all the information available. Black cells are only filled with instances from NS_{lsp}, green cells are only filled by NS_{mop}, and pink represents the cells that are filled by instances from both algorithms. White cells are those which are not filled by any method.

5.4 Summary

This chapter demonstrates the generalisation capability of the NS_{ls} and NS_{mo} methods. In particular, the performance-based descriptor approaches of these methods might allow researchers to easily switch domains.

What emerges from the results reported here is that both methods are able to generate discriminatory instances for a portfolio of TSP solvers. The results from experimenting with NS_{lsp} suggest that the method is able to differentiate the two primary solvers in the portfolio: Greedy and 2-Opt-based solvers. Strong evidence of that behaviour is presented in Figure 5.3, which provides a representation of multiple NS_{lsp} runs where the only difference is the value of ϕ to calculate the fitness of the instances. Although the instances are located very close to each other, two

well-differentiated clusters appear in the plot: a cluster of Greedy instances and another with 2-Opt instances. The lack of space between clusters may be indicative of the adjustments in the *ps* calculation and lower threshold values required to obtain discriminatory instances.

Turning now to the space coverage evaluation of the instances, we note some interesting findings. Previous work on the KP domain [90] suggested that there is a strong correlation between ϕ and the space coverage regarding U. In fact, the results from Section 5.2.1 confirm that finding by generalising the above fact to another domain of application. Table 5.2 and Figure 5.4 show that larger ϕ values tend to generate less spaced sets of instances for the TSP domain. Interestingly enough, the larger coverage is obtained when $\phi = 0.15$ instead of $\phi = 0.0$, as may be expected.

It should be noted that there are significant differences in the location of the instances over the performance space between NS_{lsp} and NS_{mop}. While NS_{lsp} exhibits a clear separation between sets of instances, NS_{mop} shows a more disordered representation. There is a large cluster of instances in the left-hand side of the space which contains a combination of Greedy and 2-Opt-based-solver instances. However, there are two well-separated clusters of instances for NN-2-Opt. This means that NS_{mop} is able to identify other patterns in the performance space that NS_{lsp} cannot. Note how Figure 5.8 illustrates the overlap and differences between both methods over the performance space.

Regarding the diversity in terms of the performance gap, the results (see Figure 5.5 and 5.7) indicate that even with the IL metric, both methods are able to generate diversity in terms of the performance gap.

Chapter 6

DIGNEA: A Diverse Instance Generator with NS and EAs

6.1 Contribution

This chapter introduces the DIGNEA framework, a Diverse Instance Generator with Novelty Search and Evolutionary Algorithms. DIGNEA is a C++ software program that has been developed to facilitate the generation of diverse and discriminatory instances for optimisation domains for the research community. The software is a generalisable C++ framework that is simultaneously capable of generating a set of instances that are diverse with respect to different search spaces (features or performance descriptors from Section 3.1.1) *and* that exhibit discriminatory but diverse performance with respect to a portfolio of solvers, where diversity, in this case, refers to variation in the magnitude of the performance gap. To achieve this, DIGNEA utilises NS_{ls} and NS_{mo} as its main components to evolve instances for any optimisation domain. A DIGNEA run requires specifying a domain, a portfolio of solvers, defining a descriptor, and a target algorithm belonging to the portfolio for whose performance the instances will be biased. Thus, a user would typically run DIGNEA once for each target algorithm in their chosen portfolio. As a result, a single DIGNEA run can potentially generate multiple instances that are diverse with respect to either features or performance and, at the same time, better suited to the chosen target in comparison to any other algorithm in the portfolio. Note that due to the stochastic nature of the methods, it is possible for some executions to result in an empty set of instances. The software not only has been used in a previous work [90], but also in [92].

It is worth mentioning at this point that DIGNEA is offered as a C++ project to build with CMake [1] from a Github repository. However, in order to facilitate the user experience, it is also distributed as a Docker image so it can be easily ported to different platforms. The image contains all the dependencies and source code of DIGNEA [2].

The remainder of this chapter is divided as follows. First, the motivation behind the development of DIGNEA is discussed. Then, an in-depth description of the software is provided by detailing the architecture and functionalities of DIGNEA. After that, an illustrative example of how to configure, run, and what type of results one can expect from DIGNEA is provided. Finally, the impact, conclusions, and future lines of work on the software are discussed.

6.2 Motivation

While most of the work in the field has focused on generating difficult instances in different domains [95, 106, 120], recent research has focused on generating instances that are maximally discriminative with respect to a portfolio of solvers proposed for a specific domain; for instance, maximising the performance gap between a target and other solvers for the Bin Packing Problem, Travelling Salesman Problem (TSP) and Knapsack Problem (KP) domains [2, 9, 66, 108]. However, most of these approaches do not include explicit mechanisms to generate instances that are diverse with respect to the feature space – they focus only on generating instances that are diverse with respect to solver performance. The work of Smith-Miles does attempt to generate space-filling instances, i.e., in unexplored regions of the feature space, but it only generates ten instances per run and needs to be repeatedly run at each point in the space where an instance is required. By contrast, alternative proposals intended to generate instances across domains rely on instances that can be solved by any solver in the portfolio [141], or even on a completely random generation process by drawing values from statistical distributions [30, 47, 106, 115, 128, 130].

DIGNEA is a generalisable C++ framework that is capable of either generating instances that are diverse with respect to a feature-space defined by a user, or of generating instances that are diverse with respect to a performance vector relating to a pre-defined portfolio, which implicitly also promotes diversity in the feature space. On

[1] https://cmake.org/.

[2] The Docker image can be accessed through: https://hub.docker.com/r/dignea/dignea

the other hand, generating diverse instances with respect to the performance vector defines the *performance space* of the portfolio used. The set of performance values with respect to a user-pre-defined portfolio is known as its performance descriptor. Unlike the feature descriptor, the performance descriptor of an instance is completely dependent on the algorithms that shape the portfolio.

A DIGNEA run requires the specification of a target algorithm belonging to the portfolio. A single run generates multiple instances that are diverse with respect to either features or performance and, at the same time, better suited to the chosen target in comparison to any other algorithm in the portfolio. Therefore, a user would typically run DIGNEA once for each target algorithm in their chosen portfolio.

6.3 Software Description

DIGNEA is written in modern C++ combining template-based types with creational design patterns that allow users to extend the framework to their needs.

6.3.1 Software Architecture

Interconnections between DIGNEA types, classes, and modules are defined by inheritance, a common approach in optimisation software [81]. `AbstractSolver` is the main algorithm interface for defining new algorithms across the entire framework. Moreover, `Problem` is a class which collects the necessary information to define any optimisation problem for which the user may want to generate instances. It allows users to define different solution representations via template parameters. To solve a problem, a solution must be defined. `Solution` is a template class that represents a typical solution for an optimisation problem. It includes the variables (genotype) and the objective values (phenotype) of a given solution for a particular problem. Since DIGNEA covers a range of pre-defined solution types, defining one's own custom solution type might be optional.

`Search` is used to create ad-hoc improvement methods to generate new solutions from a starting set of solutions. This type gives users the opportunity to create Memetic Algorithms [34], a category of EAs that includes domain-specific knowledge via custom operators such as local searches. To improve the user experience in DIGNEA, two creational design patterns were considered in its design: *builders and factories*. Builders are used to instantiate algorithms, experiments, Evolutionary Instance Generator (EIG), and Multi-objective Evolutionary Instance Generator (MOEIG) algorithms. Note that

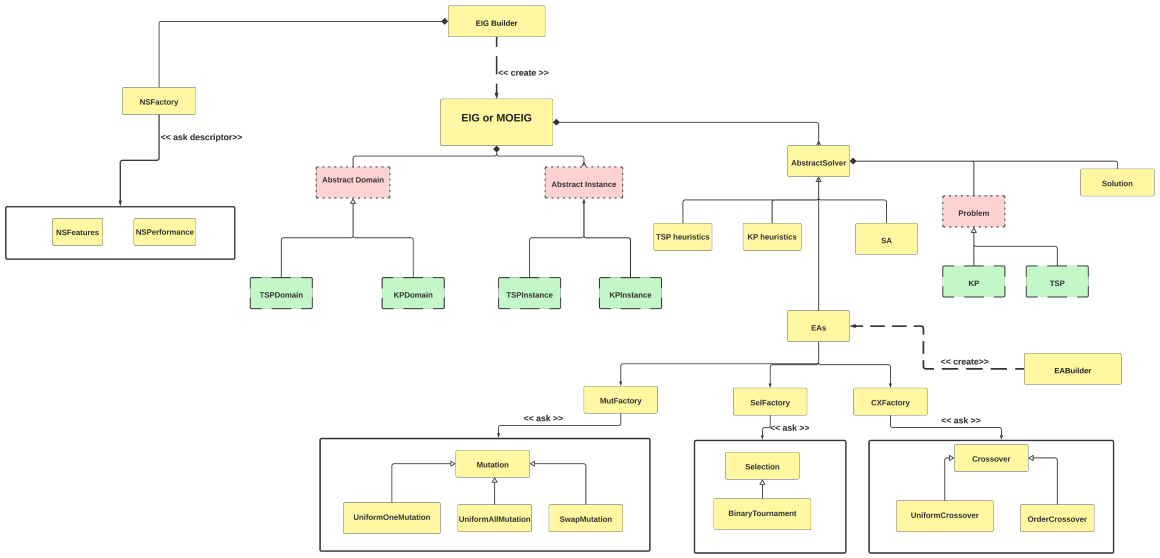


Figure 6.1 Relationship of DIGNEA main components for instance generation. Red rectangles with dotted lines represent those classes that must be extended to specify new domains. At the same time, green rectangles with dashed lines refer to the custom types for the KP or TSP domain. Yellow rectangles with straight lines are the base types used in DIGNEA that users do not necessarily need to modify.

EIG and MOEIG are the *class-name* equivalences for NS_{ls} and NS_{mo} respectively. Factories allow variation operators and other components of algorithms to be created on the fly.

From the previous building blocks, `EIG`, `AbstractDomain`, `AbstractInstance`, as well as two NS descriptor types, `NSFeatures` and `NSPerformance` were written. `EIG`, and `MOEIG` are the main instance generation components of DIGNEA. They implement different NS approaches to generate sets of diverse and discriminatory instances for any optimisation problem. `EIG` implements the NS_{ls} algorithm described in Section 3.2, and `MOEIG` is based on the multi-objective approach NS_{mo} from Section 3.3. `AbstractDomain` defines an instance generation domain for both generators, `EIG`, and `MOEIG`. Concretely, it defines a domain to generate instances for an optimisation problem P that has been previously defined in DIGNEA (an object of class `Problem`). Note that to generate instances for a domain, the optimisation problem itself P must be defined in order to solve the instances during the evolutionary process of `EIG` and `MOEIG`. `AbstractInstance` is a solution in the `AbstractDomain` domain, i.e., it represents an instance for the optimisation problem P. It includes all the information to construct an actual instance for optimisation problem P.

Figure 6.1 shows the relationship among the main instance generation components in DIGNEA. For example, to instantiate an object of type `EIG` or `MOEIG`, users must specify the following components: an NS descriptor type, such as `NSFeatures` or `NSPerformance`, created through factory `NSFactory`; a domain to generate instances for (`AbstractDomain`), such as `KPDomain`; the representation of an instance (`AbstractInstance`), such as `KPInstance`; and a portfolio of algorithms for which instances are going to be produced (`AbstractSolver`), such as `EA` or Simulated Annealing (class `SA`).

Classes `AbstractDomain` and `AbstractInstance` are completely dependent since they may include ad-hoc operations, specific attributes, and a particular representation. Therefore, to specify a new domain in DIGNEA, users *must* define at least:

1. An optimisation problem `Problem`
2. A specific domain `AbstractDomain`
3. The representation of an instance of the problem `AbstractInstance`.[3]

Colour codes and different types of lines in Figure 6.1 reflect the inheritance and extension needs in DIGNEA. Red rectangles with dotted lines represent the base classes that must be extended to specify new domains. Those in green with dashed lines are the custom types for the domains provided with the framework, and yellow rectangles with straight lines are the base types that users do not necessarily have to modify.

6.3.2 Software Functionalities

The principal contribution of DIGNEA is to offer to researchers fully generalisable software to generate instances in any optimisation domain. These instances could be then used for instance space representation, instance characterisation via hand-designed features, automate designed features, algorithm design and selection, or even parameter tuning evaluation for algorithms across domains. Moreover, since the generation of a set of instances involves the resolution of those instances with several algorithms, DIGNEA could also be used as an optimisation framework to separately validate the instances generated.

The software is not only extendable from the domain point of view, but also from the portfolio and novelty search descriptors. From the portfolio perspective, even though

[3]To see the full documentation and examples of DIGNEA, check the documentation at Github: https://dignea.github.io/.

the software is mainly conceived to use EAs, any other non-EA algorithms could also be included. The current version of DIGNEA includes the following components for instance generation:

- Solvers: Evolutionary Algorithms (`EA` objects), such as First Improve, Generational, Steady-State and Parallel Generational [89]; a Simulated Annealing approach (`SA`), four deterministic heuristics for the KP (`KPHeuristics`), as well as three deterministic heuristics for the TSP (`TSPHeuristics`).
- Novelty Search descriptor types: Novelty-Search by Features (NSFeatures) and Novelty-Search by Portfolio Performance (NSPerformance).
- Domains: Knapsack Problem (classes `KPDomain`, `KPInstance` and `KP`), Travelling Salesman Problem (classes `TSPDomain`, `TSPInstance` and `TSP`).
- Builders and Factories: `EIGBuilder`, a builder of `EIG` and `MOEIG` objects to generate instances; `EABuilder`, a builder of EA configurations; `NSFactory`, an NS descriptor factory; `CXFactory`, a crossover operator factory; `MutFactory`, a mutation operator factory; and a parent selection operator factory `SelFactory`.
- An instance printer class to generate domain-dependent instance files using the *insertion operator*.[4] This operator must be defined when creating a new domain through the extension of class `AbstractDomain`.

Regarding the NS descriptors, `NSFeatures` allows searching for diversity in a pre-defined feature space of the domain. Thus, users must define the set of features that characterise an instance in the domain and how to compute them inside the classes extending the `AbstractDomain`. Alternatively, `NSPerformance` searches for diversity in the portfolio performance space. Here, we search for instances that are diverse with respect to the performance of the solvers without considering any other information. `NSPerformance` is a suitable option for domains where the features are difficult to define or computationally expensive to calculate. It is relevant to remark that considering the above NS descriptors, diversity can be calculated using three different distance metrics: Euclidean, Manhattan, and Hamming. For further detail, refer to the NS proposal [79].

The evolutionary process performed in DIGNEA to generate instances with either `EIG` or `MOEIG` is detailed in Figure 6.2. Once the specific domain and the portfolio of

[4] C++ documentation: https://en.cppreference.com/w/cpp/io/basic_ostream/operator_ltlt2

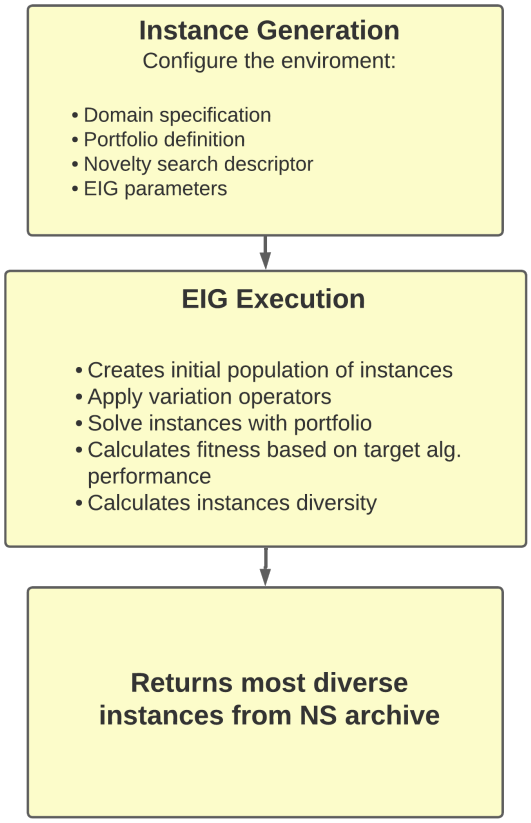


Figure 6.2 Program flowchart of an instance generation process in DIGNEA.

solvers are configured, the evolutionary process begins. First, a random population of instances is created and evaluated. After that, for G generations, the instances are evolved by following the classical EA scheme. The variation operators (crossover and mutation) are applied before evaluating all the instances with each algorithm in the portfolio. After solving all instances with each algorithm in the portfolio, the performance score ps, novelty score s, and a fitness value, if necessary, must be assigned to each instance. For example, when running `EIG`)Here we use the term fitness from the evolutionary computation field to define how suitable an instance is at a specific point during the evolutionary process. Particularly, the fitness is calculated as a linear weighted combination of two values: the performance score and the novelty score (see Equation 3.3). However, as described in Section 3.3, `MOEIG` does not uses a fitness value to generate diverse and discriminatory instances. Instead, ps and s are treated as separate objectives to maximise. For more detail about the calculation of ps and s, refer to Section 3.1.2 and 3.1.1. Also, note that the pseudo-code for the evaluations and other procedures of `EIG` and `MOEIG` are detailed in Chapter 3.

```
// Portfolio configuration
auto reps = 1;
vector<unique_ptr<AbstractSolver>> portfolio;
portfolio.push_back(make_unique<Default>());
portfolio.push_back(make_unique<MPW>());
portfolio.push_back(make_unique<MaP>());
portfolio.push_back(make_unique<MiW>());

// KP instances parameters
auto upperBound = 1000;
auto instanceSize = 100;

// EIG Parameters
auto generations = 1000;
auto nInstances = 10;
auto cxRate = 0.8;
auto mutRate = 1.0 / instanceSize;
auto multiobjective = false;
auto phi = 0.85;

// Novelty Search parameters
auto threshold = 3;
auto k = 3;
auto distance = make_unique<Euclidean<float>>();

// Domain
auto domain = make_unique<KPDomain>(instanceSize, 1, nInstances,
                    1, upperBound, 1, upperBound);
```

Figure 6.3 Parameter settings for a DIGNEA run

6.4 Illustrative Example

In this section, an example of how DIGNEA can be used to generate instances in the KP domain is provided. The example illustrates the software configuration, execution and results to obtain sets of instances similar to those analysed in Chapter 4. To do so, we only need to define the parameters for the domain, the portfolio of solvers, and the particular NS descriptor to run the experiment. The documentation available at the Github repository[5] provides complete tutorials tp create new domains, algorithms and run instance generation experiments.

Notice that to simplify the example, some template parameters and other irrelevant code have been omitted.

[5]The Github repository of DIGNEA can be accessed through: https://github.com/DIGNEA/dignea.

```
unique_ptr<EIG> generator =
   EIGBuilder::create(multiobjective)
       .toSolve(move(instKP))
       .with()
       .weights(phi, 1.0 - phi)
       .portfolio(portfolio)
       .repeating(reps)
       .withSearch(NSType::Features, move(distance), threshold, k)
       .with()
       .crossover(CXType::Uniform)
       .mutation(MutType::UniformOne)
       .selection(SelType::Binary)
       .withMutRate(mutRate)
       .withCrossRate(cxRate)
       .populationOf(nInstances)
       .runDuring(generations);

generator->run(); // Runs EIG to generate KP instances

auto instances = generator->getResults(); // Collect the results
```

Figure 6.4 C++ source code fragment to generate KP instances in DIGNEA

In our example, we are using a portfolio of four deterministic KP heuristics [108] to evolve KP instances based on the performance of those solvers. The portfolio is defined as a C++ vector of unique pointers to `AbstractSolver` objects. Note that the order in which the vector is populated is extremely important. The solver *in the first position*, i.e., Default, located at position zero in the vector, will be considered as the target solver, and therefore the instances generated will be tailored so this solver outperforms the remaining solvers in the portfolio when solving those particular instances. Figure 6.3 illustrates the creation of the KP heuristic portfolio and another parameter setting to run the example. Considering the KP domain, we set the number of items in the instances to $N = 50$, and the bounds $w_i, p_i \forall i \in N$ to be in the range $[1, 1000]$. Note that the size of the instance must be set twice as large as the expected outcome. For instance, if we want to generate KP instances with $N = 50$ items, the instance size defined for `EIG` or `MOEIG` must be 100. The reason behind this decision is based on how the instances are represented in memory when evolved in DIGNEA. For the KP domain, item i is separated into w_i and p_i so the recombination and mutation operators can act on profits or weights independently, instead of modifying the entire item at the same time. For more detail, refer to Figure 3.3 and Section 3.1.1. Another important parameter to consider is the multi-objective boolean flag passed to `EIGBuilder`. This parameter determines whether `EIGBuilder` will create an

instance of `EIG` or `MOEIG`. For instance, in this example the value is false, meaning the algorithm created is an `EIG` instance. Moreover, the number of objectives for the domain must be set accordingly. This restriction is related to the inner creation of objects of type `AbstractInstance` and how the evaluation process differs from `EIG` and `MOEIG`. For example, when working with `EIG`, the number of objectives for domains, such as `KPDomain`, must be set to one. Each domain may define the different parameters required for a proper configuration of the environment. However, every domain must initialise the expected parameters for an `AbstractDomain`, which are the following: dimension (N as twice as much of the actual instance size), number of objectives, and the number of constraints (usually zero). Apart from that, `KPDomain` requires the definition of the number of instances to generate (same value as the population size of `EIG`), lower and upper bounds for the weights and profits of the items in the instances. Therefore, Figure 6.3 line 28 defines a KP domain where:

- The dimension of the instances for `EIG` is set to 100, generating KP instances with $N = 50$ items.
- The number of objectives is set to 1. We are using `EIG`.
- The number of instances to generate is equal to 10.
- The bounds (low, up) for each weight in the resulting instances are: (1, 1000).
- The bounds (low, up) for each profit in the resulting instances are: (1, 1000).

After that, we use `EIGBuilder` to create a unique pointer to an `EIG` object with all the configuration required to generate a set of instances for the KP domain. Notice that `EIGBuilder` facilitates the initialisation process by using the necessary factories, i.e., *withSearch* uses the `NSFactory` type to create a new NS descriptor object of type `NSFeatures` with the remaining arguments. The above is illustrated in the fragment of C++ code shown in Figure 6.4. This source code fragment shows that the builder pattern is used to create a unique pointer to an `EIG` object. The methods are self-explanatory. Calling the method `run` starts the evolutionary process, and the method `getResults` provides the set of instances at the end of the execution.

Figure 6.5 shows a Javascript Object Notation (JSON) file generated from the experiment detailed in this section. The JSON files usually contain all the information from the algorithm such as `EIG`, the domain (`KPDomain` in our example), and a *set* of solutions. The set contains the set of diverse instances generated by the EIG. Each instance contains its novelty score, performance score, fitness, and actual instance information. It may also contain its feature values if defined.

```
    { "algorithm": {
            "portfolio": [
             {  "isTarget": true,  "name": "Default KP"
             },
             {  "isTarget": false, "name": "MPW KP"
             },
             {  "isTarget": false, "name": "MaP KP"
             },
             {  "isTarget": false, "name": "MiW KP"
             }
        ],
    (...) // Other relevant information
    "set": {
        "0": { // First instance
            "novelty": 6938.35498046875,
            "features": {
                "Q": 23678.0,
                "avg_eff": 0.7300000190734863,
                "max_p": 991.0,
                "max_w": 985.0,
                "mean": 538.2100219726563,
                "min_p": 8.0,
                "min_w": 48.0,
                "std": 298.83905029296875
            },
            "fitness": 116.80326843261719,
            "n_vars": 100, // Inst. definition starts here
            "capacity": 23678,
            "profits": [
                // 50 integers, one for each p_i
            ],
            "weights": [
                // 50 integers, one for each w_i
            ]
        },
        (...) // More instances
        ,
        "n_solutions": 37,
    },
}}
```


Figure 6.5 JSON file with the results from the instance generation experiment. This type of file is directly provided by DIGNEA.

Applying a simple data analysis procedure of scalarisation and dimensionality reduction can provide a clear visualisation of the instances produced, as demonstrated through the entire Chapter 4.

6.5 Impact

Since the ASP was defined back in 1976, researchers have struggled to address this computationally expensive problem. Traditionally, solving the ASP for a domain involves at least a three-step procedure. First, we start by generating and gathering a large number of instances to generate a dataset. Then, after defining a portfolio of solvers, a considerably expensive computational experiment is required, i.e., solving each instance in the dataset with every solver in the portfolio. Thereafter, a selection mechanism must be applied to associate each instance to the solver that yields the best performance. This process is very time-consuming and prone to human mistakes when switching from one step to the next. Besides, in most cases, the instance generation process is performed randomly and is not guaranteed to be representative of the domain or emphasise the strengths/weaknesses of the solvers.

DIGNEA was designed to facilitate this process by combining the previous three steps into one single procedure. The workflow of the software not only ensures that the instances generated are correctly labelled to the best-performing solver in the portfolio, but it also allows researchers to define how diverse they want the instances to be with respect to one another. Moreover, the modular and template-based architecture of DIGNEA provides researchers a straightforward way to switch domains and include their own solvers. Although DIGNEA allows the one-step instance generation process to be simplified, the above is still a computationally expensive task. The solvers in the portfolio and the number of repetitions to perform on each generation must be selected carefully. From our own experience, running DIGNEA with EAs and a large number of evaluations to perform may involve several days, if not weeks, of computational work. However, the software supports MPI parallelism of the experiments and, as a result, it can be run in HPC systems to speed-up the process. For instance, DIGNEA has been deployed and executed correctly in various HPC systems such as Archer2[6] and TeideHPC.[7]

DIGNEA can be of great assistance to gain more insight into problems, their instances, and how these instances share the feature and performance spaces with the aim of better designing them. Furthermore, DIGNEA can be applied to pursue new research questions, for example, about instance space representation, instance characterisation via hand-designed features, automate designed features, algorithm design and selection, or even parameter tuning evaluation for algorithms across domains.

[6]Archer2 website: https://www.archer2.ac.uk/
[7]TeideHPC website: https://teidehpc.iter.es/

6.6 Conclusions and Future Lines of Work

The main objective of this chapter is to present the software DIGNEA, a Diverse Instance Generator with Novelty Search and Evolutionary Algorithms. DIGNEA is defined using generic types and a modular architecture to facilitate its extension to other domains and portfolios. Although the current version of the software contains the required types for generating instances for the KP and TSP domain, work is in progress to include more domains, such as Bin-Packing or the Menu Planning Problem. The functionalities of DIGNEA have been widely proven in the previous chapters of this thesis. Results have shown that it is able to generate better instances with respect to space coverage, novelty, and fitness [90] in comparison to previous approaches that only considered the generation of random instances to maximise the performance gap among solvers [108, 128]. Furthermore, the software has been released as open source to the research community [92].

Moreover, future lines of work on DIGNEA may include performance optimisation to speed-up the generation process, and interfacing the C++ framework with Python libraries to not only promote the use of other scientific libraries in the process, but also to reach a wider audience that may not be comfortable with C++ programming. Moreover, a pure Python re-implementation of DIGNEA could be also considered; however, the viability of this line in terms of computational performance must be studied.

Chapter 7

Conclusions

Generating sets of instances is a common task among computer scientists in the optimisation field. Traditionally, statistical or random methods have been applied to produce hard-to-solve instances or 'benchmarks' that can be used to analyse the strengths and weaknesses of state-of-the-art algorithms in the field. Recently, numerous proposals in the field of instance generation have focused on applying evolutionary methods such as EAs for biased instance generation. However, most do not consider diversity a relevant factor, thus producing high-redundant sets of instances and poor coverage of the instance spaces. As discussed in Chapters 1 and 2, producing instance sets that are as diverse as possible is desirable in many contexts, not only in the analysis of state-of-the-art algorithms. The early stages of this thesis focused on one of those contexts, the ASP, and soon realised the importance of the instance generation process in the optimisation domains.

The aim of this thesis was to investigate the generation of diverse but also discriminatory instances in optimisation domains by means of (1) proposing NS methods to generate diverse and discriminatory instances with respect to a portfolio of solvers, (2) evaluating the impact of problem formulation in the resulting set, (3) evaluating its generalisation across optimisation domains and (4) developing open-source software in C++ to facilitate the application of the methods to other researchers in the field. Furthermore, this thesis tried to bring attention to the generation of diverse and discriminatory instances and somehow bridge the gap between the two traditional trends in the literature: the generation of pure diverse instances without even considering the performance of different algorithms, and the generation of hard-to-solve instances for state-of-the-art solvers that tend to produce homogeneous sets of instances. The methods proposed in this thesis are able to take advantage of the best of both worlds

and thus produce diverse sets of instances for which one solver outperforms others in a portfolio.

7.1 Key Results

This section is devoted to answering the research questions presented in Chapter 1 and to discussing other important findings from the experimental assessments presented in Chapters 4 and 5.

- **Question 1: To what extent can a Novelty Search algorithm be used to generate diverse and discriminatory instances that aim to provide uniform coverage of the descriptor space for a portfolio of algorithms?** The experimental evaluation addressed in this thesis was designed to evaluate the extent to which NS methods can be applied to generate diverse and discriminatory instances. Although most of the work is focused on the KP domain, Chapter 5 also provided some insights into the TSP domain. As a result of this research, two major NS algorithms are proposed: NS_{ls} and NS_{mo}. The former uses a fitness function that drives the evolutionary process, which consists of a linear combination of novelty s and performance ps scores using a weighted-sum aggregation, while the latter introduces a multi-objective methodology where each objective is optimised simultaneously. In addition to the above, a deeper level of instance evaluation was introduced: the instance descriptors. Two descriptors are considered: a feature-based and a performance-based descriptor. In the KP domain, the features descriptor is built with a set of eight features. On the other hand, the performance-based descriptor is defined as an M-dimensional vector with the average performance of each solver considered in a portfolio of size M. Therefore, there are four NS methods in total: NS_{lsf}, NS_{lsp} NS_{mof}, and NS_{mop}.

  Throughout Chapters 4 and 5 of this thesis, both NS methods (NS_{ls} and NS_{mo}) and descriptors have been evaluated to generate diverse and discriminatory instances for KP and TSP domains. The results from this thesis, in conjunction with previous works [87, 90], suggest that both are able to succeed in this task, although there are a few limitations. First, both methods contain a significant number of parameters that can take on a vast range of different values, making the parameterisation of the algorithms considerably more difficult when switching domains or descriptors. Another important finding is the unexpectedly low scores in terms of unique instances in the KP domain (see Tables 4.3 and 4.6). This

finding was also reported in previous work [87], where the mean unique instance ratio remains around 30 per cent. However, for the TSP domain, the analysis of the impact of ϕ shows that NS_{lsp} is able to produce sets of instances with extremely low levels of redundancy for $\phi > 0.3$. Finally, even though a standard EA approach (such as [108]) seems to produce sets with higher unique ratios than NS_{ls} variants, both NS variants outperformed these approaches in the KP domain in terms of space coverage U (covering larger areas of the features and performance spaces) and performance gap(see Table 4.2). By contrast, NS_{mo} variants outperformed the NSGA-II algorithm to generate diverse and discriminatory instances in both unique instances (in terms of $Unique_s$ and $Unique_p$) and space coverage U (see Table 4.5).

- **Question 2: How diverse are the instances evolved for each target with respect to the performance gap, i.e., the magnitude of the difference between the performance of the winning solver and the remaining solvers in the portfolio?**
  One of the most interesting outcomes from the experimental evaluation of this thesis is the diversity in terms of the performance gap. The first signs of diversity with these metrics appeared when evaluating a portfolio of EAs in the KP domain [90]. Throughout Chapters 4 and 5, several figures provided a quantitative analysis of the performance gap distribution for each run. It is important to bear in mind that the distribution of values depends on the portfolio and the metric used to evaluate it. For instance, KP results from Chapter 4 present significantly larger values than results from the TSP domain, Chapter 5, due to different metrics; i.e., profit versus IL. Moreover, note that it is much more difficult to generate discriminatory instances for the two heuristics MPW and Def than for MaP and MiW in the KP domain. It is not clear whether this is due to the fact that these heuristics are intrinsically weak compared to the other two, and hence there are very few cases in which they outperform the other methods, or whether the algorithm fails to locate them. Even though such behaviour must be expected in NS_{lsp} and NS_{mop}, results from NS_{lsf}, NS_{mof} and [90] reveal that the methods are able to generate instances that are diverse in terms of the performance difference when using a feature-based descriptor as well. A likely explanation is that searching for diversity in the feature space may produce different descriptors, and therefore instances, which tend to be easier or harder to solve for a solver.

- **Question 3: To what extent can a Novelty Search algorithm using either feature or performance descriptor provide useful information of the opposite search space? Are the Novelty Search methods with a performance descriptor able to uniformly distribute the instances in the feature space and vice versa?**
  In sections 4.1.3 and 4.2.3, we evaluated the capabilities of NS_{ls} and NS_{mo} to provide useful information about the foreign space, with respect to the descriptor utilised in each run. Moreover, each method was compared with a standard EA method as a base model; i.e., [108] for NS_{ls} and an NSGA-II for NS_{mo}. In addition, we provide qualitative and quantitative results of the distribution of instances across both spaces for each method. In Section 4.1.3, Figure 4.5 illustrated that while [108] clustered all the instances in the centre of both spaces, NS_{lsf} and NS_{lsp} were able to better distribute the instances in their respective spaces. Moreover, Table 4.2 demonstrates that each NS_{ls} method outperforms its opponents in their 'own' space with respect to the U metric; i.e., NS_{lsf} scored $U = 0.5883$ in the features space while NS_{lsp} reached $U = 0.6426$ in the performance space. Besides, [108] obtained significantly lower coverage values in both spaces. Additionally, Section 4.2.3 presents an identical analysis for NS_{mo} approaches. Results from Figure 4.10 and Table 4.5 exhibited similar behaviour to NS_{ls}. Interestingly enough, although NS_{mo} was able to generate larger sets of instances per run, neither method was able to score better coverage values than NS_{ls}. NS_{mof} scored $U = 0.5681$ and NS_{mop} scored $U = 0.5437$ in their respective spaces. This behaviour could be an indication that the NS_{mo} may require an in-depth parameter tuning evaluation to generate slightly more spread sets of instances. Furthermore, in this evaluation, NSGA-II scored even lower coverage values than [108], with $U < 0.35$ in both spaces.

- **Question 4: To what extent can the formulation of the instance generation problem impact the resulting sets of instances? In other words, are there substantial differences in diversity, space coverage and the total number of instances in the sets generated per run between a Novelty Search method using a single-objective approach and a Multi-objective-based Novelty Search algorithm?**
  There are some interesting outcomes that can be drawn from the experimental evaluation in Chapter 4. First, while the NS_{ls} approach was able to generate more diverse sets of instances in both spaces, NS_{mo} was able to create larger ones. However, NS_{mo} coverage scores are still competitive when compared with NS_{ls}.

Section 4.3 provides a direct comparison between NS_{ls} and NS_{mo} approaches in both spaces. The results revealed that each method was able to explore different regions of the spaces. Thus, even though there exists a slight overlap, they could be complementary. Figures 4.11 and 4.12 show that an ensemble of both approaches was able to cover more of the space than any individual method and that each one contributes towards the final collection. Therefore, there is value in using an ensemble approach. Finally, we could not conclude that the problem strongly benefits from one or another approach, but that a combination of both methods is desirable. However, in light of the results, we may suggest using NS_{ls} when more diverse sets are required, and NS_{mo} when the total amount of instances is more important. Besides, these results are only applied to the KP domain and should not be extrapolated to all domains

- **Question 5: To what extent can Novelty Search methods be generalised to other optimisation domains when relying on performance-based descriptors?**
  The methods presented here are generalisable to any combinatorial optimisation domain; particularly, by using NS_{lsp} or NS_{mop}, which does not require an intensive problem domain analysis and is a much more straightforward method in comparison to NS_{lsf} and NS_{mof}. To illustrate this, Chapter 5 demonstrated the application of both NS_{lsp} and NS_{mop} methods in the TSP domain. The TSP is a well-known domain for which several feature descriptors have been proposed [9, 10, 122]. However, some of these features can be difficult or computationally expensive to calculate. For this reason, it is a suitable domain to evaluate the ability to generalise of the NS methods presented here. We evaluated both methods using a portfolio of three TSP heuristics to create TSP instances with 50 nodes. The results demonstrated that NS_{lsp} was not only able to proficiently differentiate the two principal heuristics (Greedy from 2-Opt types) in the portfolio (see Figure 5.3), but it also obtained similar coverage scores and performance gap diversity to the KP domain. By contrast, NS_{mop} was not able to group the instances per solvers like NS_{lsp}. Figure 5.6 shows how most of the instances are located in the left bottom of the performance space, with small clusters of NN-2-Opt instances in the right and top areas. Nevertheless, NS_{mop} was able to obtain competitive results in terms of space coverage U and performance gap diversity. Furthermore, Figure 5.8 illustrated that, since there exists a clear difference between both methods, an ensemble could provide some benefits in terms of space coverage and diversity. In conclusion, the methods proposed here

can be applied to more domains than just the KP. The results from Chapter 5 demonstrated that both methods are able to produce high-quality results when using a performance-based descriptor. Besides, we encourage the application of these descriptors to facilitate a much more straightforward transition to other domains. However, if time and computational resources are no object, defining a feature descriptor could also be desirable.

Another interesting outcome from the research in this thesis is the correlation between ϕ and the space coverage U in NS_{lsp}. This behaviour was previously investigated in the KP domain with NS_{lsf} and an EA-based portfolio [90]. Section 5.2.1 corroborates the findings from [90], i.e., *larger ϕ values tend to create less diverse sets of instances*. In fact, Figure 5.4 shows a strong negative correlation between ϕ and U. Pearson's Correlation Coefficient (r) supports that statement with $r = -0.9234$.

7.2 Future Work

In terms of future work, there are several interesting lines that may be addressed. First, both NS_{ls} and NS_{mo} only consider the performance difference between two solvers when calculating the performance score *ps* of an instance, target and the maximum performance among others. Future work can be directed towards updating the *ps* calculation to consider more than one solver, such as [12]. Therefore, the performance score will reflect the difference between all solvers in the portfolio instead of between the target solver and the best solver among others. Second, other interesting lines of research can include evaluating the NS methods with diverse portfolios and generating larger instances.

We have collected many KP instances as a result of the experimental evaluation performed over these years. Hence, future work may include the design of Machine Learning models to solve the ASP in the KP domain. Moreover, this work may benefit from the previously mentioned lines. Obtaining instances with various sizes and biased to different solvers, more than just deterministic heuristics and EAs [90], could potentiate the ability to perform algorithm selection by providing new augmented datasets to train machine-learning-based classifiers.

Considering the software DIGNEA, future work may be directed to include new domains such as the BPP and new solvers to existing and new domains. Therefore, we aim to extend the capabilities of the software to facilitate the application of the methods to the research community. In addition, performance optimisation to speed up the generation process, and interfacing the C++ framework with Python libraries

to reach a wider audience that may not be comfortable with C++ programming are other interesting projects.

Finally, another future line of research is to transfer the methods to a real-world problem. This thesis started investigating the Menu Planning Problem in school cafeterias from Tenerife [88, 91]. We considered a novel constrained multi-objective formulation for the MPP, where the cost of the menus and the level of repetition are the two objectives that have to be minimised simultaneously. Moreover, some studies suggest that the MPP can be reduced to a MDKP problem [116]. However, there are slight differences in terms of the instances between KP and MPP. An instance in the MPP is represented by a set of meals prepared with one or more ingredients (each ingredient has a collection of nutritional facts), which has an associated cost. Besides, the amount of ingredients required to prepare a meal also affects the final cost. Since it is impossible to create ingredients that do not exist in the real world, there are fewer options to evolve diverse instances in the MPP domain. The main alternatives are to (1) initialise the instances with meals from previously collected recipes and evolve them by changing the amount of each ingredient, and (2) create custom meals with ingredients extracted from specialised databases such as FoodData [1] or Food Composition Data [2]. Furthermore, the formulation of the MPP domain in DIGNEA may also introduce a domain-specific set of constraints to avoid undesirable outcomes.

7.3 Publications Resulting from the Research of this Thesis

The following journal articles and conference papers, listed in chronological order, were published during the period of study resulting in this thesis.

7.3.1 Journal Articles

- Marrero, A., Segredo, E., León, C. and Hart, E. 2023. DIGNEA: A Tool to Generate Diverse and Discriminatory Instance Suites for Optimisation Domains. SoftwareX 22.
- Marrero, A., Segredo, E., León, C. and Segura, C. 2020. A Memetic Decomposition-Based Multi-Objective Evolutionary Algorithm Applied to a Constrained Menu Planning Problem. Mathematics 8, 1-18.

[1] https://fdc.nal.usda.gov/

[2] https://www.efsa.europa.eu/es/microstrategy/food-composition-data

7.3.2 International Conferences

- Marrero, A., Segredo, E., Hart, E., Bossek, J., and Neumann, A. 2023. Generating diverse and discriminatory knapsack instances by searching for novelty in variable dimensions of feature-space. In Proceedings of the Genetic and Evolutionary Computation Conference (GECCO '23). Association for Computing Machinery, New York, NY, USA, 312–320.

- Marrero, A., Segredo, E., León, C., and Hart, E. 2022. A Novelty-Search Approach to Filling an Instance-Space with Diverse and Discriminatory Instances for the Knapsack Problem. In Parallel Problem Solving from Nature – PPSN XVII: 17th International Conference, PPSN 2022, Dortmund, Germany, September 10–14, 2022, Proceedings, Part I. Springer-Verlag, Berlin, Heidelberg, 223–236.

- Marrero, A., Segredo, E., and León, C. 2021. A parallel genetic algorithm to speed up the resolution of the algorithm selection problem. In Proceedings of the Genetic and Evolutionary Computation Conference Companion (GECCO '21). Association for Computing Machinery, New York, NY, USA, 1978–1981.

- Marrero, A., Segredo, E., and León, C. 2019. On the automatic planning of healthy and balanced menus. In Proceedings of the Genetic and Evolutionary Computation Conference Companion (GECCO '19). Association for Computing Machinery, New York, NY, USA, 71–72.

7.3.3 Spanish National Conferences

- Marrero, A., Segredo, E. and León, C. 2018. Combinación De Computación Evolutiva Y Aprendizaje Automatizado En La Resolución De Problemas De Optimización. In XVIII Conferencia de la Asociación Española para la Inteligencia Artificial (CAEPIA 2018): Avances en Inteligencia Artificial. 23-26 de octubre de 2018 Granada, España, Asociación Española para la Inteligencia Artificial (AEPIA), 1361-1366.

Bibliography

[1] Charu C Aggarwal, Alexander Hinneburg, and Daniel A Keim. 2001. On the Surprising Behavior of Distance Metrics in High Dimensional Space. In *Database Theory — ICDT 2001*, Jan den Bussche and Victor Vianu (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 420–434.

[2] Mohamad Alissa, Kevin Sim, and Emma Hart. 2019. Algorithm Selection Using Deep Learning without Feature Extraction. In *Proceedings of the Genetic and Evolutionary Computation Conference* (Prague, Czech Republic) *(GECCO '19)*. Association for Computing Machinery, New York, NY, USA, 198–206.

[3] R. Andonov, V. Poirriez, and S. Rajopadhye. 2000. Unbounded knapsack problem: Dynamic programming revisited. *European journal of operational research* 123, 2 (2000), 394–407.

[4] Enrico Angelelli, Renata Mansini, and M. Grazia Speranza. 2010. Kernel search: A general heuristic for the multi-dimensional knapsack problem. *Computers & operations research* 37, 11 (2010), 2017–2026.

[5] Federico Antonello, Piero Baraldi, Enrico Zio, and Luigi Serio. 2022. A novelty-based multi-objective evolutionary algorithm for identifying functional dependencies in complex technical infrastructures from alarm data. *Environment systems & decisions* 42, 2 (2022), 177–188.

[6] Tiziano Bacci and Sara Nicoloso. 2021. *On the Benchmark Instances for the Bin Packing Problem with Conflicts*. Springer International Publishing, Cham, 171–179.

[7] Tapan P. Bagchi, Jatinder N.D. Gupta, and Chelliah Sriskandarajah. 2006. A review of TSP based approaches for flowshop scheduling. *European Journal of Operational Research* 169, 3 (2006), 816–854.

[8] Jon Bentley. 1992. Fast Algorithms for Geometric Traveling Salesman Problems. *INFORMS J. Comput.* 4 (1992), 387–411.

[9] Jakob Bossek, Pascal Kerschke, Aneta Neumann, Markus Wagner, Frank Neumann, and Heike Trautmann. 2019. Evolving Diverse TSP Instances by Means of Novel and Creative Mutation Operators. In *Proceedings of the 15th ACM/SIGEVO Conference on Foundations of Genetic Algorithms* (Potsdam, Germany) *(FOGA '19)*. Association for Computing Machinery, New York, NY, USA, 58–71.

[10] Jakob Bossek and Frank Neumann. 2022. Exploring the Feature Space of TSP Instances Using Quality Diversity. In *Proceedings of the Genetic and Evolutionary Computation Conference* (Boston, Massachusetts) *(GECCO '22)*. Association for Computing Machinery, New York, NY, USA, 186–194.

[11] Jakob Bossek and Heike Trautmann. 2016. Understanding Characteristics of Evolved Instances for State-of-the-Art Inexact TSP Solvers with Maximum Performance Difference. In *Proceedings of the XV International Conference of the Italian Association for Artificial Intelligence on Advances in Artificial Intelligence - Volume 10037 (AI*IA 2016)*. Springer-Verlag, Berlin, Heidelberg, 3–12.

[12] Jakob Bossek and Markus Wagner. 2021. Generating Instances with Performance Differences for More than Just Two Algorithms. In *Proceedings of the Genetic and Evolutionary Computation Conference Companion* (Lille, France) *(GECCO '21)*. Association for Computing Machinery, New York, NY, USA, 1423–1432.

[13] Ilhem Boussaïd, Julien Lepagnot, and Patrick Siarry. 2013. A survey on optimization metaheuristics. *Information Sciences* 237 (2013), 82–117.

[14] V. Boyer, M. Elkihel, and D. El Baz. 2009. Heuristics for the 0–1 multidimensional knapsack problem. *European journal of operational research* 199, 3 (2009), 658–664.

[15] Sally C. Brailsford, Chris N. Potts, and Barbara M. Smith. 1999. Constraint satisfaction problems: Algorithms and applications. *Eur. J. Oper. Res.* 119, 3 (1999), 557–581.

[16] Kurt M Bretthauer and Bala Shetty. 2002. The nonlinear knapsack problem – algorithms and applications. *European Journal of Operational Research* 138, 3 (2002), 459–472.

[17] Edgar Buchanan, Léni K Le Goff, Wei Li, Emma Hart, Agoston E Eiben, Matteo De Carlo, Alan F Winfield, Matthew F Hale, Robert Woolley, Mike Angus, et al. 2020. Bootstrapping artificial evolution to design robots for autonomous fabrication. *Robotics* 9, 4 (2020), 106.

[18] Valentina Cacchiani, Manuel Iori, Alberto Locatelli, and Silvano Martello. 2022. Knapsack problems — An overview of recent advances. Part I: Single knapsack problems. *Computers & operations research* 143 (2022), 105692.

[19] Konstantinos Chatzilygeroudis, Antoine Cully, Vassilis Vassiliades, and Jean-Baptiste Mouret. 2021. Quality-Diversity Optimization: A Novel Branch of Stochastic Optimization. In *Black Box Optimization, Machine Learning, and No-Free Lunch Theorems*, Panos M Pardalos, Varvara Rasskazova, and Michael N Vrahatis (Eds.). Springer International Publishing, Cham, 109–135.

[20] Omar Cheikhrouhou and Ines Khoufi. 2021. A comprehensive survey on the Multiple Traveling Salesman Problem: Applications, approaches and taxonomy. *Computer science review* 40 (2021), 100369.

[21] Tinkle Chugh, Karthik Sindhya, Jussi Hakanen, and Kaisa Miettinen. 2019. A survey on handling computationally expensive multiobjective optimization problems with evolutionary algorithms. *Soft computing (Berlin, Germany)* 23, 9 (2019), 3137–3166.

[22] Grit Claßen, Arie M.C.A. Koster, and Anke Schmeink. 2015. The multi-band robust knapsack problem—A dynamic programming approach. *Discrete optimization* 18 (2015), 123–149.

[23] G. A. Croes. 1958. A Method for Solving Traveling-Salesman Problems. *Operations Research* 6, 6 (1958), 791–812.

[24] Giuseppe Cuccu and Faustino Gomez. 2011. When Novelty Is Not Enough. In *Applications of Evolutionary Computation*, Cecilia Di Chio, Stefano Cagnoni, Carlos Cotta, Marc Ebner, Anikó Ekárt, Anna I. Esparcia-Alcázar, Juan J. Merelo, Ferrante Neri, Mike Preuss, Hendrik Richter, Julian Togelius, and Georgios N. Yannakakis (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 234–243.

[25] Antoine Cully and Jean-Baptiste Mouret. 2013. Behavioral Repertoire Learning in Robotics. In *Proceedings of the 15th Annual Conference on Genetic and Evolutionary Computation* (Amsterdam, The Netherlands) *(GECCO '13)*. Association for Computing Machinery, New York, NY, USA, 175–182.

[26] Miguel Cárdenas-Montes. 2018. Creating hard-to-solve instances of travelling salesman problem. *Applied soft computing* 71 (2018), 268–276.

[27] Tobias (1884-1956) Dantzig. 1954. *Number: the language of science* (4th ed., rev. and augm.. ed.). The Macmillan Limited, New York.

[28] Kalyanmoy Deb and Kalyanmoy Deb. 2014. *Multi-objective Optimization*. Springer US, Boston, MA, 403–449.

[29] Kalyanmoy Deb, Amrit Pratap, Sameer Agarwal, and T. Meyarivan. 2002. A Fast and Elitist Multiobjective Genetic Algorithm: NSGA-II. *IEEE Transactions on Evolutionary Computation* 6 (4 2002), 182–197. Issue 2.

[30] Maxence Delorme, Manuel Iori, and Silvano Martello. 2016. Bin packing and cutting stock problems: Mathematical models and exact algorithms. *European Journal of Operational Research* 255, 1 (2016), 1–20.

[31] Ioannis C. Demetriou and Panos M. Pardalos. 2019. *No Free Lunch Theorem: A Review*. Vol. 145. Switzerland: Springer International Publishing AG, Switzerland, 57–82.

[32] Stephane Doncieux, Alban Laflaquière, and Alexandre Coninx. 2019. Novelty Search: A Theoretical Perspective. In *Proceedings of the Genetic and Evolutionary Computation Conference* (Prague, Czech Republic) *(GECCO '19)*. Association for Computing Machinery, New York, NY, USA, 99–106.

[33] Ke-Lin Du and M. N. S. Swamy. 2016. *Search and Optimization by Metaheuristics: Techniques and Algorithms Inspired by Nature* (1st ed.). Birkhäuser Basel, Dordrecht. 327–336 pages.

[34] Agoston E. Eiben and James E. Smith. 2015. *Introduction to Evolutionary Computing*. Springer Berlin, Heidelberg, Heidelberg. 287 pages.

[35] Andries P. Engelbrecht. 2007. *Introduction to Computational Intelligence*. John Wiley & Sons, Ltd, California, USA, Chapter 1, 1–13.

[36] Guillaume Escamocher, Barry O'Sullivan, and Steven Prestwich. 2020. Generating Difficult CNF Instances in Unexplored Constrainedness Regions. *The ACM journal of experimental algorithmics* 25, 1 (Jul 04, 2020), 1–12.

[37] Absalom E. Ezugwu, Verosha Pillay, Divyan Hirasen, Kershen Sivanarain, and Melvin Govender. 2019. A Comparative Study of Meta-Heuristic Optimization Algorithms for 0 - 1 Knapsack Problem: Some Initial Results. *IEEE access* 7 (2019), 43979–44001.

[38] Evelyn Fix and J. L. Hodges. 1989. Discriminatory Analysis. Nonparametric Discrimination: Consistency Properties. *International Statistical Review / Revue Internationale de Statistique* 57, 3 (1989), 238–247.

[39] R. Fletcher. 1994. *An Overview of Unconstrained Optimization*. Springer Netherlands, Dordrecht, 109–143.

[40] Merrill M. Flood. 1956. The Traveling-Salesman Problem. *Operations research* 4, 1 (1956), 61–75. 12.

[41] L. J. Fogel, A. J. Owens, and M. J. Walsh. 1965. Intelligent decision-making through a simulation of evolution. *IEEE transactions on human factors in electronics* HFE-6, 1 (1965), 13–23.

[42] Karl Pearson F.R.S. 1901. LIII. On lines and planes of closest fit to systems of points in space. *Philosophical Magazine Series 1* 2 (1901), 559–572.

[43] B. Fuglede and F. Topsoe. 2004. Jensen-Shannon divergence and Hilbert space embedding. In *International Symposium on Information Theory, 2004. ISIT 2004. Proceedings*. IEEE, Chicago, IL, USA, 31–.

[44] G. Gallo, P. L. Hammer, and B. Simeone. 1980. *Quadratic knapsack problems*. Springer Berlin Heidelberg, Berlin, Heidelberg, 132–149.

[45] Wanru Gao, Samadhi Nallaperuma, and Frank Neumann. 2016. Feature-Based Diversity Optimization for Problem Instance Classification. In *Parallel Problem Solving from Nature – PPSN XIV*, Julia Handl, Emma Hart, Peter R. Lewis, Manuel López-Ibáñez, Gabriela Ochoa, and Ben Paechter (Eds.). Springer International Publishing, Cham, 869–879.

[46] Wanru Gao, Samadhi Nallaperuma, and Frank Neumann. 2021. Feature-based diversity optimization for problem instance classification. *Evolutionary Computation* 29, 1 (2021), 107–128.

[47] T. Gau and G. Wäscher. 1995. CUTGEN1: A problem generator for the standard one-dimensional cutting stock problem. *European Journal of Operational Research* 84, 3 (1995), 572–579.

[48] Mitsuo Gen and Lin Lin. 2014. Multiobjective evolutionary algorithm for manufacturing scheduling problems: state-of-the-art survey. *Journal of intelligent manufacturing* 25, 5 (2014), 849–866.

[49] Aurelien Geron. 2019. *Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow: Concepts, Tools, and Techniques to Build Intelligent Systems* (2nd ed.). O'Reilly Media, Inc., California 95472.

[50] Marc Goerigk and Mohammad Khosravi. 2022. Benchmarking Problems for Robust Discrete Optimization. arXiv:2201.04985 [math.OC]

[51] Marc Goerigk and Stephen J. Maher. 2020. Generating hard instances for robust combinatorial optimization. *European Journal of Operational Research* 280, 1 (2020), 34–45.

[52] Jacob Goldberger, Geoffrey E Hinton, Sam Roweis, and Russ R Salakhutdinov. 2004. Neighbourhood Components Analysis. In *Advances in Neural Information Processing Systems*, L. Saul, Y. Weiss, and L. Bottou (Eds.), Vol. 17. MIT Press, London, England.

[53] Jorge Gomes, Pedro Mariano, and Anders Lyhne Christensen. 2015. Devising Effective Novelty Search Algorithms: A Comprehensive Empirical Study. In *Proceedings of the 2015 Annual Conference on Genetic and Evolutionary Computation* (Madrid, Spain) *(GECCO '15)*. Association for Computing Machinery, New York, NY, USA, 943–950.

[54] Haipeng Guo and William H. Hsu. 2007. A machine learning approach to algorithm selection for NP-hard optimization problems: A case study on the MPE problem. *Annals of operations research* 156, 1 (2007), 61–82.

[55] Doug Hains, Darrell Whitley, and Adele Howe. 2012. Improving Lin-Kernighan-Helsgaun with Crossover on Clustered Instances of the TSP. In *Parallel Problem Solving from Nature - PPSN XII*, Carlos A. Coello Coello, Vincenzo Cutello, Kalyanmoy Deb, Stephanie Forrest, Giuseppe Nicosia, and Mario Pavone (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 388–397.

[56] Melanie Herzog, Sebastian Lotz, and Wolfgang F. Riedl. 2014. The Traveling Salesman Problem. https://www-m9.ma.tum.de/games/tsp-game/

[57] Y.C. Ho and D.L. Pepyne. 2002. Simple Explanation of the No-Free-Lunch Theorem and Its Implications. *Journal of Optimization Theory and Applications* 115 (12 2002), 549–570. Issue 3.

[58] Dorit S. Hochbaum. 1995. A Nonlinear Knapsack Problem. *Oper. Res. Lett.* 17, 3 (apr 1995), 103–110.

[59] Abdollah Homaifar, Charlene X. Qi, and Steven H. Lai. 1994. Constrained Optimization Via Genetic Algorithms. *SIMULATION* 62, 4 (1994), 242–253.

[60] Satoshi Horie and Osamu Watanabe. 1997. Hard instance generation for SAT. In *Algorithms and Computation*, Hon Wai Leong, Hiroshi Imai, and Sanjay Jain (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 22–31.

[61] Mohammad Hossin and Md Nasir Sulaiman. 2015. A review on evaluation metrics for data classification evaluations. *International journal of data mining & knowledge management process* 5, 2 (2015), 1.

[62] Stefan Hougardy and Xianghui Zhong. 2021. Hard to solve instances of the Euclidean Traveling Salesman Problem. *Mathematical programming computation* 13, 1 (2021), 51–74.

[63] Kashif Hussain, Mohd N. Mohd Salleh, Shi Cheng, and Yuhui Shi. 2019. Metaheuristic research: a comprehensive survey. *The Artificial intelligence review* 52, 4 (2019), 2191–2233.

[64] Colin Johnson, Vic Ciesielski, João Correia, and Penousal Machado. 2016. *Fitness and Novelty in Evolutionary Art*. Vol. 9596. Switzerland: Springer International Publishing AG, Switzerland, 225–240.

[65] Jorik Jooken, Pieter Leyman, and Patrick De Causmaecker. 2022. A new class of hard problem instances for the 0–1 knapsack problem. *European Journal of Operational Research* 301, 3 (2022), 841–854.

[66] Bryant Julstrom. Jul 08, 2009. Evolving heuristically difficult instances of combinatorial problems. In *Proceedings of the 11th Annual conference on Genetic and evolutionary computation (GECCO)*. Association for Computing Machinery - ACM, New York, NY, USA, 279–286.

[67] Bryant A. Julstrom. 2009. Evolving Heuristically Difficult Instances of Combinatorial Problems. In *Proceedings of the 11th Annual Conference on Genetic and Evolutionary Computation* (Montreal, Québec, Canada) *(GECCO '09)*. Association for Computing Machinery, New York, NY, USA, 279–286.

[68] André A. Keller. 2017. *Multi-Objective Optimization in Theory and Practice I: Classical Methods*. Bentham eBook, Bentham Science Publishers Executive Suite Y - 2 Building Y Saif Zone Sharjah, U.A.E. 296 pages.

[69] Andre A. Keller. 2019. *Multi-Objective Optimization in Theory and Practice II: Metaheuristic Algorithms*. Bentham Science Publishers, Bentham Science Publishers Executive Suite Y - 2 Building Y Saif Zone Sharjah, U.A.E. 310 pages.

[70] Pascal Kerschke, Holger H Hoos, Frank Neumann, and Heike Trautmann. 2019. Automated algorithm selection: Survey and perspectives. *Evolutionary computation* 27, 1 (2019), 3–45.

[71] Ines Khoufi, Anis Laouiti, and Cedric Adjih. 2019. A survey of recent extended variants of the traveling salesman and vehicle routing problems for unmanned aerial vehicles. *Drones (Basel)* 3, 3 (2019), 1–30.

[72] Jong-Hwan Kim and Hyun Myung. 1997. Evolutionary programming techniques for constrained optimization problems. , 129-140 pages.

[73] Wilhelm Kirch (Ed.). 2008. *Pearson's Correlation Coefficient*. Springer Netherlands, Dordrecht, 1090–1091.

[74] John R. Koza. 1992. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. MIT Press, Cambridge, MA, USA.

[75] Rajeev Kumar and Nilanjan Banerjee. 2006. Analysis of a Multiobjective Evolutionary Algorithm on the 0–1 knapsack problem. *Theoretical computer science* 358, 1 (2006), 104–120.

[76] Xiangjing Lai, Jin-Kao Hao, Fred Glover, and Zhipeng Lü. 2018. A two-phase tabu-evolutionary algorithm for the 0–1 multidimensional knapsack problem. *Information sciences* 436-437 (2018), 282–301.

[77] Léni K. Le Goff, Emma Hart, Alexandre Coninx, and Stéphane Doncieux. 2020. On Pros and Cons of Evolving Topologies with Novelty Search. In *Proceeedings of the ALIFE 2020: The 2020 Conference on Artificial Life (ALIFE 2022: The 2022 Conference on Artificial Life, Vol. ALIFE 2020: The 2020 Conference on Artificial Life)*, Josh Bongard, Juniper Lovato, Laurent Hebert-Dufrésne, Radhakrishna Dasari, and Lisa Soros (Eds.). MIT Press, Montréal, Canada, 423–431.

[78] Joel Lehman and Kenneth O. Stanley. 2010. Efficiently Evolving Programs through the Search for Novelty. In *Proceedings of the 12th Annual Conference on Genetic and Evolutionary Computation* (Portland, Oregon, USA) *(GECCO '10)*. Association for Computing Machinery, New York, NY, USA, 837–844.

[79] Joel Lehman and Kenneth O. Stanley. 2011. Abandoning objectives: Evolution through the search for novelty alone. *Evolutionary Computation* 19, 2 (2011), 189–222.

[80] J. K. Lenstra and A. H. G. Rinnooy Kan. 1975. Some Simple Applications of the Travelling Salesman Problem. *Operational Research Quarterly (1970-1977)* 26, 4 (1975), 717–733.

[81] Coromoto León, Gara Miranda, and Carlos Segura. 2009. METCO: a Parallel Plugin-Based Framework for Multi-Objective Optimization. *International Journal on Artificial Intelligence Tools* 18, 04 (2009), 569–588.

[82] Antonios Liapis, Georgios N. Yannakakis, and Julian Togelius. 2015. Constrained Novelty Search: A Study on Game Content Generation. *Evol. Comput.* 23, 1 (mar 2015), 101–129.

[83] Zhichao Lu, Ian Whalen, Vishnu Boddeti, Yashesh Dhebar, Kalyanmoy Deb, Erik Goodman, and Wolfgang Banzhaf. 2019. NSGA-Net: Neural Architecture Search Using Multi-Objective Genetic Algorithm. In *Proceedings of the Genetic and Evolutionary Computation Conference* (Prague, Czech Republic) *(GECCO '19)*. Association for Computing Machinery, New York, NY, USA, 419–427.

[84] Thibaut Lust and Jacques Teghem. 2010. *The Multiobjective Traveling Salesman Problem: A Survey and a New Approach*. Vol. 272. Berlin, Heidelberg: Springer Berlin Heidelberg, Berlin, Heidelberg, 119–141.

[85] Thibaut Lust and Jacques Teghem. 2012. The multiobjective multidimensional knapsack problem: a survey and a new approach. *International Transactions in Operational Research* 19, 4 (2012), 495–520.

[86] William G. Macready and David H. Wolpert. 1996. What makes an optimization problem hard? *Complexity* 1, 5 (1996), 40–46.

[87] Alejandro Marrero, Eduardo Segredo, Emma Hart, Jakob Bossek, and Aneta Neumann. 2023. Generating Diverse and Discriminatory Knapsack Instances by Searching for Novelty in Variable Dimensions of Feature-Space. In *Proceedings of the Genetic and Evolutionary Computation Conference* (Lisbon, Portugal) *(GECCO '23)*. Association for Computing Machinery, New York, NY, USA, 312–320.

[88] Alejandro Marrero, Eduardo Segredo, and Coromoto Leon. 2019. On the automatic planning of healthy and balanced menus. In *GECCO 2019 Companion - Proceedings of the 2019 Genetic and Evolutionary Computation Conference Companion*. Association for Computing Machinery, Inc, Prague, Czech Republic, 71–72.

[89] Alejandro Marrero, Eduardo Segredo, and Coromoto Leon. 2021. A Parallel Genetic Algorithm to Speed up the Resolution of the Algorithm Selection Problem. In *Proceedings of the Genetic and Evolutionary Computation Conference Companion* (Lille, France) *(GECCO '21)*. Association for Computing Machinery, New York, NY, USA, 1978–1981.

[90] Alejandro Marrero, Eduardo Segredo, Coromoto León, and Emma Hart. 2022. A Novelty-Search Approach to Filling an Instance-Space with Diverse and Discriminatory Instances for the Knapsack Problem. In *Parallel Problem Solving from Nature – PPSN XVII*. Springer International Publishing, Cham, 223–236.

[91] Alejandro Marrero, Eduardo Segredo, Coromoto León, and Carlos Segura. 2020. A Memetic Decomposition-Based Multi-Objective Evolutionary Algorithm Applied to a Constrained Menu Planning Problem. *Mathematics* 8, 11 (2020), 18 pages.

[92] Alejandro Marrero, Eduardo Segredo, Coromoto León, and Emma Hart. 2023. DIGNEA: A tool to generate diverse and discriminatory instance suites for optimisation domains. *SoftwareX* 22 (2023), 101355.

[93] Silvano Martello, David Pisinger, and Paolo Toth. 1999. Dynamic Programming and Strong Bounds for the 0-1 Knapsack Problem. *Management Science* 45, 3 (Mar 01, 1999), 414–424.

[94] Radek Matousek. 2018. Stochastic Heuristics for Knapsack Problems. In *Advances in Intelligent Systems and Computing*. Advances in Intelligent Systems and Computing, Vol. 837. Springer International Publishing AG, Switzerland, 157–166.

[95] Krzysztof Michalak. 2021. Generating hard inventory routing problem instances using evolutionary algorithms.

[96] Seyedali Mirjalili. 2018. *Evolutionary Algorithms and Neural Networks: Theory and Applications*. Studies in Computational Intelligence, Vol. 780. Springer International Publishing AG, Cham.

[97] Melanie Mitchell. 1998. *An Introduction to Genetic Algorithms*. MIT Press, Cambridge, MA, USA.

[98] Jean-Baptiste Mouret. 2011. Novelty-Based Multiobjectivization. In *New Horizons in Evolutionary Robotics*, Vol. 341. Berlin, Heidelberg: Springer Berlin Heidelberg, Berlin, Heidelberg, 139–154.

[99] Jean-Baptiste Mouret and Jeff Clune. 2015. Illuminating search spaces by mapping elites.

[100] Jean-Baptiste Mouret and Stéphane Doncieux. 2012. Encouraging Behavioral Diversity in Evolutionary Robotics: an Empirical Study. *Evolutionary Computation* 20, 1 (2012), 91–133.

[101] Mario A Muñoz and Kate Smith-Miles. 2020. Generating new space-filling test instances for continuous black-box optimization. *Evolutionary computation* 28, 3 (2020), 379–404.

[102] Christian Nilsson. 2003. Heuristics for the Traveling Salesman Problem. *Linkoping University* 38 (2003), 00085–9.

[103] Christos Papalitsas and Theodore Andronikos. 2019. Unconventional GVNS for Solving the Garbage Collection Problem with Time Windows. *Technologies (Basel)* 7, 3 (2019), 61.

[104] David Pisinger. 1995. An expanding-core algorithm for the exact 0–1 knapsack problem. , 175 pages.

[105] David Pisinger. 2000. A Minimal Algorithm for the Bounded Knapsack Problem. *INFORMS journal on computing* 12, 1 (2000), 75–82.

[106] David Pisinger. 2005. Where are the hard knapsack problems? *Computers and Operations Research* 32, 9 (2005), 2271–2284.

[107] David Pisinger. 2007. The quadratic knapsack problem—a survey. *DISCRETE APPLIED MATHEMATICS* 155, 5 (2007), 623–648.

[108] Luis Fernando Plata-González, Ivan Amaya, José Carlos Ortiz-Bayliss, Santiago Enrique Conant-Pablos, Hugo Terashima-Marín, and Carlos A. Coello Coello. 2019. Evolutionary-based tailoring of synthetic instances for the Knapsack problem. *Soft Computing* 23, 23 (2019), 12711–12728.

[109] Justin K Pugh, Lisa B Soros, and Kenneth O Stanley. 2016. Quality diversity: A new frontier for evolutionary computation. *Frontiers in Robotics and AI* 3 (2016), 40.

[110] Justin K. Pugh, L. B. Soros, Paul A. Szerlip, and Kenneth O. Stanley. 2015. Confronting the Challenge of Quality Diversity. In *Proceedings of the 2015 Annual Conference on Genetic and Evolutionary Computation* (Madrid, Spain) *(GECCO '15)*. Association for Computing Machinery, New York, NY, USA, 967–974.

[111] César Rego, Dorabela Gamboa, Fred Glover, and Colin Osterman. 2011. Traveling salesman problem heuristics: Leading methods, implementations and latest advances. *European Journal of Operational Research* 211, 3 (2011), 427–441.

[112] John R. Rice. 1976. The Algorithm Selection Problem. *Advances in Computers* 15, C (1976), 65–118.

[113] Aiying Rong, José Rui Figueira, and Kathrin Klamroth. 2012. Dynamic programming based algorithms for the discounted 0–1 knapsack problem. *Applied mathematics and computation* 218, 12 (2012), 6921–6933.

[114] Hans-Paul Paul Schwefel. 1993. *Evolution and Optimum Seeking: The Sixth Generation*. John Wiley & Sons, Inc., USA.

[115] P. Schwerin and G. Wäscher. 1997. The Bin-Packing Problem: A Problem Generator and Some Numerical Experiments with FFD Packing and MTP. *International Transactions in Operational Research* 4, 5-6 (1997), 377–389.

[116] Barbara K. Seljak. 2009. Computer-based dietary menu planning. *Journal of food composition and analysis* 22, 5 (Aug 01, 2009), 414–420.

[117] David K Smith. 1991. *Dynamic programming : a practical introduction* (1st pub.. ed.). Ellis Horwood, New York [etc.].

[118] Kate Smith-Miles, Davaatseren Baatar, Brendan Wreford, and Rhyd Lewis. 2014. Towards objective measures of algorithm performance across instance space. *Computers & Operations Research* 45 (2014), 12–24.

[119] Kate Smith-Miles and Simon Bowly. 2015. Generating new test instances by evolving in instance space. *Computers and Operations Research* 63 (2015), 102–113.

[120] Kate Smith-Miles, Jeffrey Christiansen, and Mario A. Muñoz. 2021. Revisiting where are the hard knapsack problems? via Instance Space Analysis. *Computers & Operations Research* 128 (2021), 105184.

[121] Kate Smith-Miles and Leo Lopes. 2012. Measuring instance difficulty for combinatorial optimization problems. *Computers & operations research* 39, 5 (May 2012), 875–889.

[122] Kate Smith-Miles, Jano van Hemert, and Xin Yu Lim. 2010. Understanding TSP Difficulty by Learning from Evolved Instances. In *Learning and Intelligent Optimization*, Christian Blum and Roberto Battiti (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 266–280.

[123] Kate A. Smith-Miles. 2009. Cross-Disciplinary Perspectives on Meta-Learning for Algorithm Selection. *ACM Comput. Surv.* 41, 1, Article 6 (jan 2009), 25 pages.

[124] Paul Szerlip, Gregory Morse, Justin Pugh, and Kenneth Stanley. 2015. Unsupervised Feature Learning through Divergent Discriminative Feature Accumulation. *Proceedings of the AAAI Conference on Artificial Intelligence* 29, 1 (Feb. 2015), 7 pages.

[125] Igor V. Tetko, David J. Livingstone, and Alexander I. Luik. 1995. Neural network studies. 1. Comparison of overfitting and overtraining. *Journal of chemical information and computer sciences* 35, 5 (1995), 826–833.

[126] Alan Turing. 2004. Intelligent Machinery (1948). In *The Essential Turing*. Oxford University Press, London, UK.

[127] Levent Türkler, Taner Akkan, and Lütfiye Özlem Akkan. 2022. Usage of Evolutionary Algorithms in Swarm Robotics and Design Problems. *Sensors (Basel, Switzerland)* 22, 12 (2022), 4437.

[128] Markus Ullrich, Thomas Weise, Abhishek Awasthi, and Jörg Lässig. 2018. A generic problem instance generator for discrete optimization problems.

[129] Neil Urquhart and Emma Hart. 2018. Optimisation and Illumination of a Real-World Workforce Scheduling and Routing Application (WSRP) via Map-Elites. In *Parallel Problem Solving from Nature – PPSN XV*, Anne Auger, Carlos M. Fonseca, Nuno Lourenço, Penousal Machado, Luís Paquete, and Darrell Whitley (Eds.). Springer International Publishing, Cham, 488–499.

[130] E. Vallada, R. Ruiz, and J. M. Framinan. 2015. New hard benchmark for flowshop scheduling problems minimising makespan. *European Journal of Operational Research* 240, 3 (2015), 666–677.

[131] Jano I. van Hemert. 2006. Evolving Combinatorial Problem Instances That Are Difficult to Solve. *Evol. Comput.* 14, 4 (dec 2006), 433–462.

[132] LingFeng Wang, Kay Chen Tan, and Chee Meng Chew. 2006. *Evolutionary robotics: from algorithms to implementations*. World Scientific series in robotics and intelligent systems, Vol. 28. World Scientific Publishing Co. Pte. Ltd, Singapore.

[133] Shimin Wang. 2022. Optimization and Simulation of Electrical Wiring Path Based on Ant Colony Parallel Algorithm. In *2021 International Conference on Big Data Analytics for Cyber-Physical System in Smart City*, Mohammed Atiquzzaman, Neil Yen, and Zheng Xu (Eds.). Springer Singapore, Singapore, 267–272.

[134] Geoffrey I. Webb, Eamonn Keogh, Risto Miikkulainen, Risto Miikkulainen, and Michele Sebag. 2010. No-Free-Lunch Theorem. *Encyclopedia of Machine Learning* 1 (2010), 721–721. Issue 1.

[135] Christophe Wilbaut, Said Hanafi, and Said Salhi. 2008. A survey of effective heuristics and their application to a variety of knapsack problems. *IMA journal of management mathematics* 19, 3 (2008), 227–244.

[136] Gerhard J. Woeginger. 2003. *Exact Algorithms for NP-Hard Problems: A Survey.* Springer Berlin Heidelberg, Berlin, Heidelberg, 185–207.

[137] Yu Wu. 2021. A survey on population-based meta-heuristic algorithms for motion planning of aircraft. *Swarm and evolutionary computation* 62 (2021), 100844.

[138] Enrico Zardini, Davide Zappetti, Davide Zambrano, Giovanni Iacca, and Dario Floreano. 2021. Seeking Quality Diversity in Evolutionary Co-Design of Morphology and Control of Soft Tensegrity Modular Robots. In *Proceedings of the Genetic and Evolutionary Computation Conference* (Lille, France) *(GECCO '21)*. Association for Computing Machinery, New York, NY, USA, 189–197.

[139] Jing Zhang. 2009. Natural Computation for the Traveling Salesman Problem. In *2009 Second International Conference on Intelligent Computation Technology and Automation*, Vol. 1. IEEE, Changsha, China, 366–369.

[140] Ming Zhong and Bo Xu. 2017. A Developmental Evolutionary Algorithm for 0-1 Knapsack Problem. In *Cloud Computing and Security*, Xingming Sun, Han-Chieh Chao, Xingang You, and Elisa Bertino (Eds.). Springer International Publishing, Cham, 849–854.

[141] Özgür Akgün, Nguyen Dang, Ian Miguel, András Z. Salamon, and Christopher Stone. 2019. Instance Generation via Generator Instances. In *Principles and Practice of Constraint*, Thomas Schiex and Simon de Givry (Eds.). Springer International Publishing, Cham, 3–19.

Appendix A

Deterministic KP heuristics

A.1 Default KP Heuristic

Algorithm 9: Default KP Heuristic

Input : Q, N, X

$q = 0$;
$p = 0$;
$x = \emptyset$;
$i = 0$;
while $q \leq Q$ *and* $i \leq N$ **do**
    if $X_{i_w} + q \leq Q$ **then**
        $x = x \cup \{i\}$;
        $q = q + X_{i_q}$;
        $p = p + X_{i_p}$;
    end
    $i = i + 1$;
end
return x, p, q;

A.2 Maximum Profit KP Heuristic

Algorithm 10: Maximum Profit (MaP) KP Heuristic

Input : Q, N, X

$q = 0$;
$p = 0$;
$x = \emptyset$;
$X = \text{sort\_by\_profit\_reverse}(X)$;
$i = 0$;
while $q \leq Q$ *and* $i \leq N$ **do**
    if $X_{i_w} + q \leq Q$ **then**
        $x = x \cup \{i\}$;
        $q = q + X_{i_q}$;
        $p = p + X_{i_p}$;
    end
    $i = i + 1$;
end
return x, p, q;

A.3 Maximum Profit per Weight KP Heuristic

Algorithm 11: Maximum Profit per Weight (MPW) KP Heuristic

Input : Q, N, X

$q = 0$;
$p = 0$;
$x = \emptyset$;
$X = \text{sort\_by\_efficiency}(X)$;
$i = 0$;
while $q \leq Q$ *and* $i \leq N$ **do**
  if $X_{i_w} + q \leq Q$ **then**
    $x = x \cup \{i\}$;
    $q = q + X_{i_q}$;
    $p = p + X_{i_p}$;
  end
  $i = i + 1$;
end
return x, p, q;

A.4 Minimum Weight KP Heuristic

Algorithm 12: Minimum Weight (MiW) KP Heuristic

Input : Q, N, X

```
1  q = 0;
2  p = 0;
3  x = ∅;
4  X = sort_by_weight(X);
5  i = 0;
6  Done = False;
7  while not Done do
8      if X_{i_w} + q ≤ Q then
9          x = x ∪ {i} ;
10         q = q + X_{i_q};
11         p = p + X_{i_p};
12         i = i + 1;
13     else
14         Done = True;
15     end
16 end
17 return x, p, q;
```

Appendix B

Figures from ϕ Parameter Tuning Experiment for TSP Domain

B.1 Distribution of NS_{lsp} Instances across Performance Space

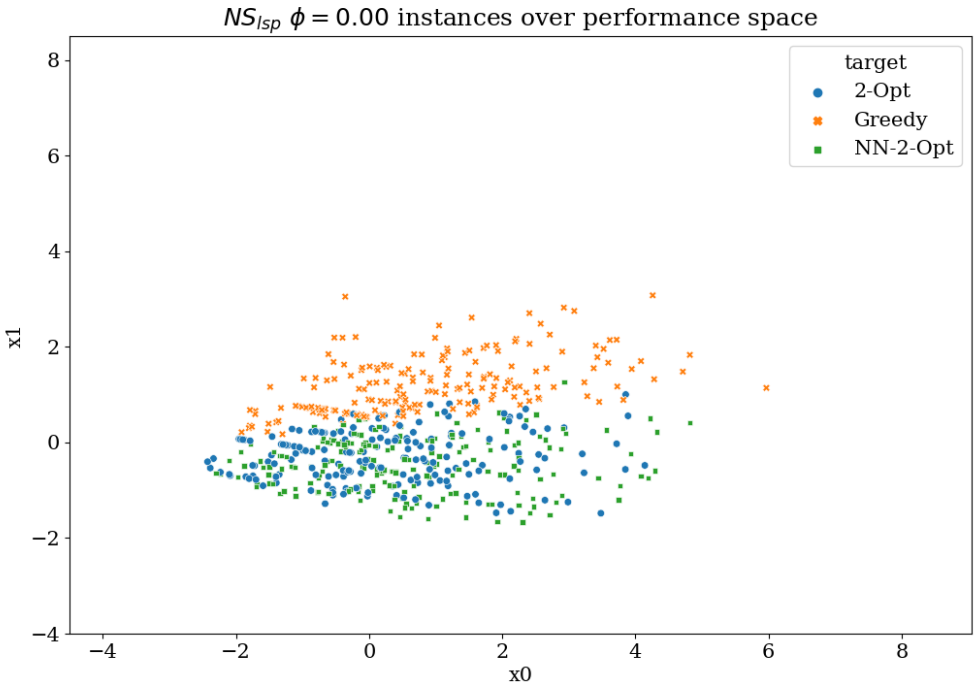


Figure B.1 Instance representation in a 2D space after applying PCAs to all the instances generated by NS_{lsp} setting $\phi = 0.0$. The colours and symbols reflect the 'winning' solver for each instance. Blue dots are the instances generated for 2-Opt, orange crosses Greedy, and green squares are the instances for NN-2-Opt.

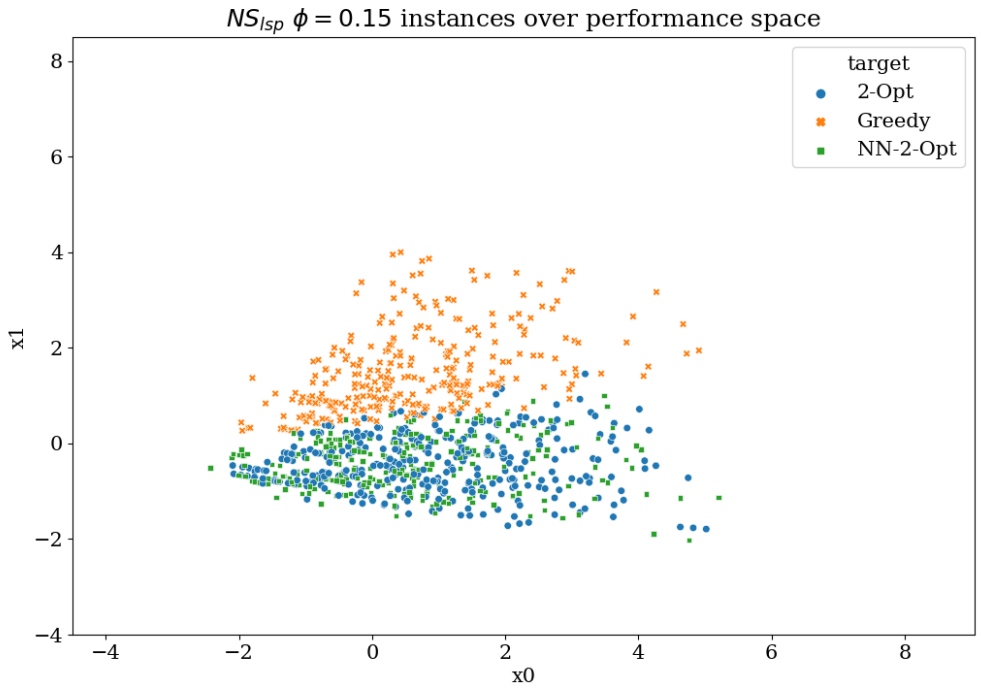


Figure B.2 Instance representation in a 2D space after applying PCAs to all the instances generated by NS_{lsp} setting $\phi = 0.15$. The colours and symbols reflect the 'winning' solver for each instance. Blue dots are the instances generated for 2-Opt, orange crosses Greedy, and green squares are the instances for NN-2-Opt.

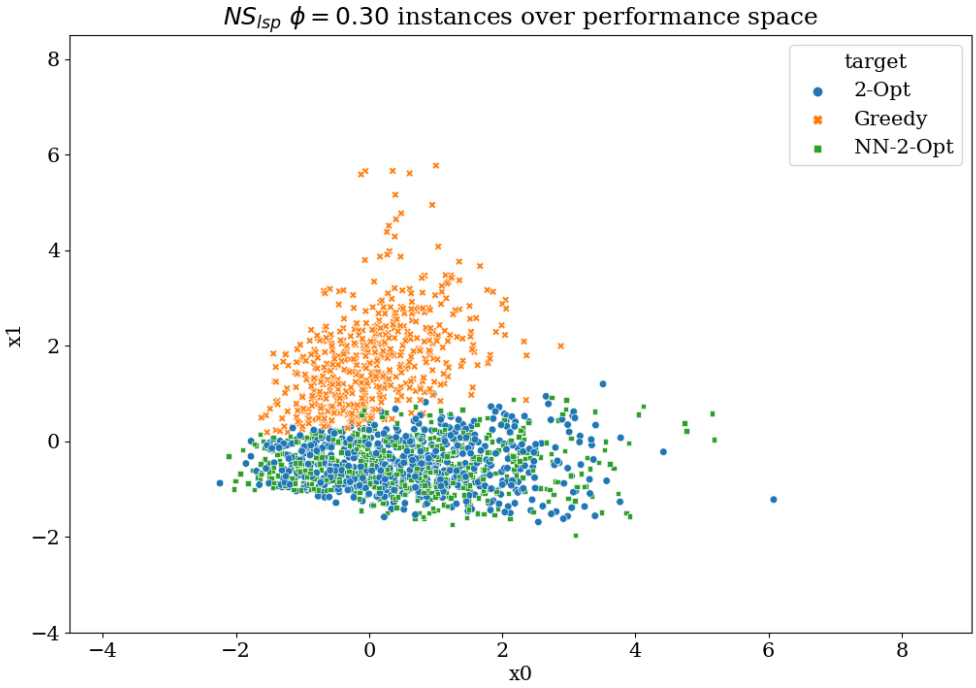


Figure B.3 Instance representation in a 2D space after applying PCAs to all the instances generated by NS_{lsp} setting $\phi = 0.30$. The colours and symbols reflect the 'winning' solver for each instance. Blue dots are the instances generated for 2-Opt, orange crosses Greedy, and green squares are the instances for NN-2-Opt.

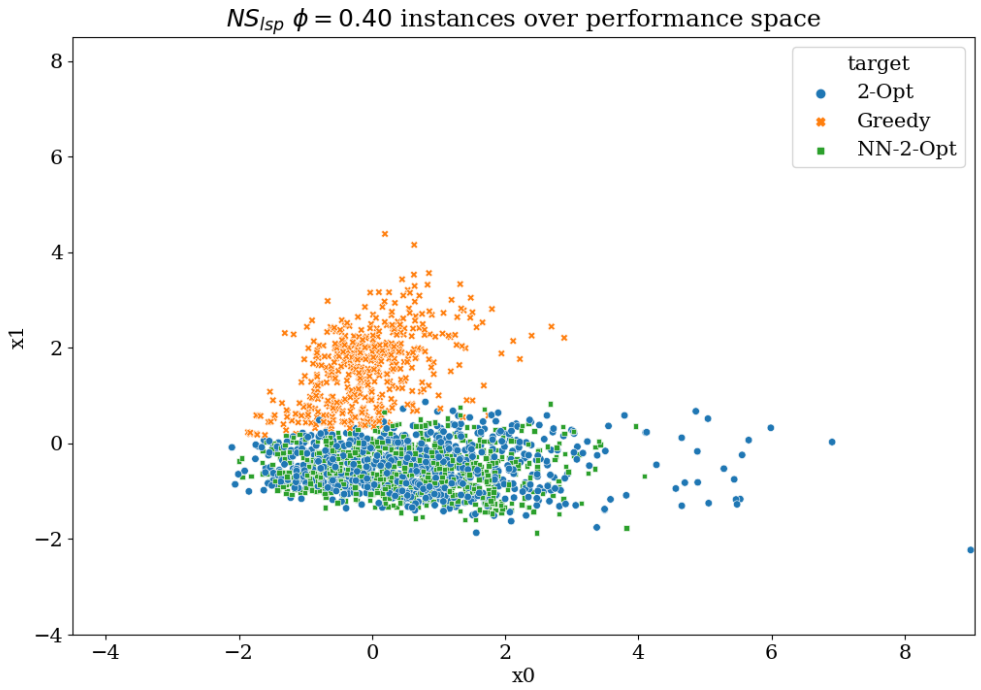


Figure B.4 Instance representation in a 2D space after applying PCAs to all the instances generated by NS_{lsp} setting $\phi = 0.40$. The colours and symbols reflect the 'winning' solver for each instance. Blue dots are the instances generated for 2-Opt, orange crosses Greedy, and green squares are the instances for NN-2-Opt.

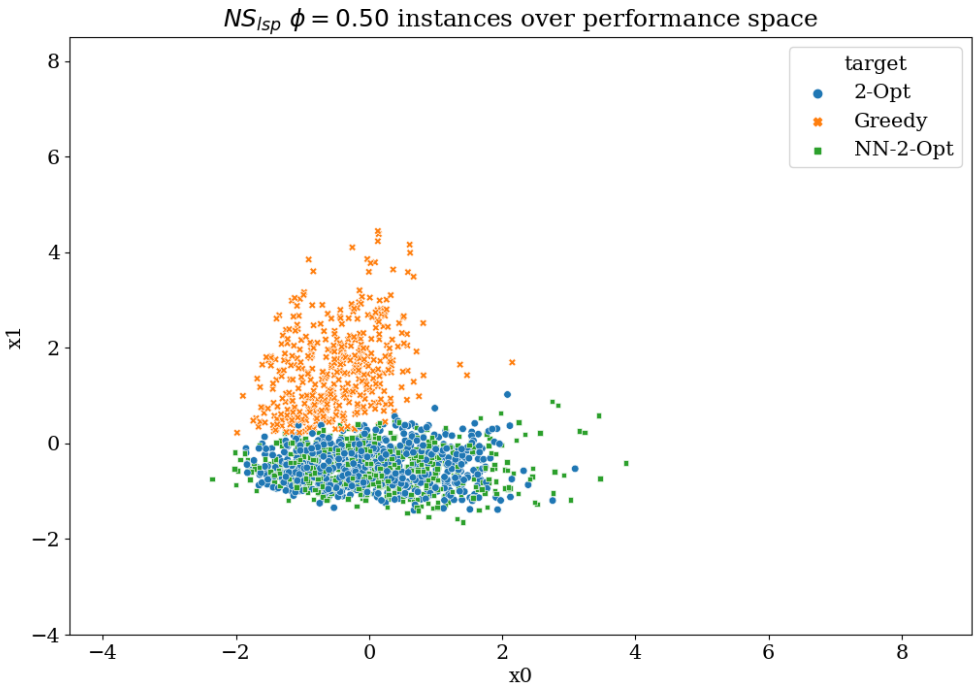


Figure B.5 Instance representation in a 2D space after applying PCAs to all the instances generated by NS_{lsp} setting $\phi = 0.50$. The colours and symbols reflect the 'winning' solver for each instance. Blue dots are the instances generated for 2-Opt, orange crosses Greedy, and green squares are the instances for NN-2-Opt.

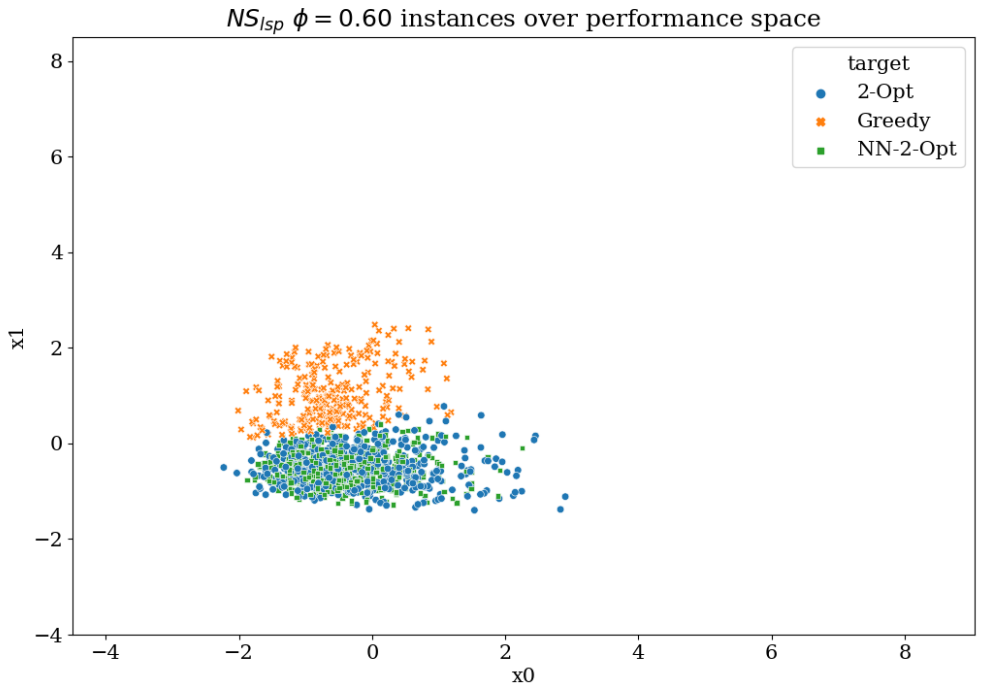


Figure B.6 Instance representation in a 2D space after applying PCAs to all the instances generated by NS_{lsp} setting $\phi = 0.60$. The colours and symbols reflect the 'winning' solver for each instance. Blue dots are the instances generated for 2-Opt, orange crosses Greedy, and green squares are the instances for NN-2-Opt.

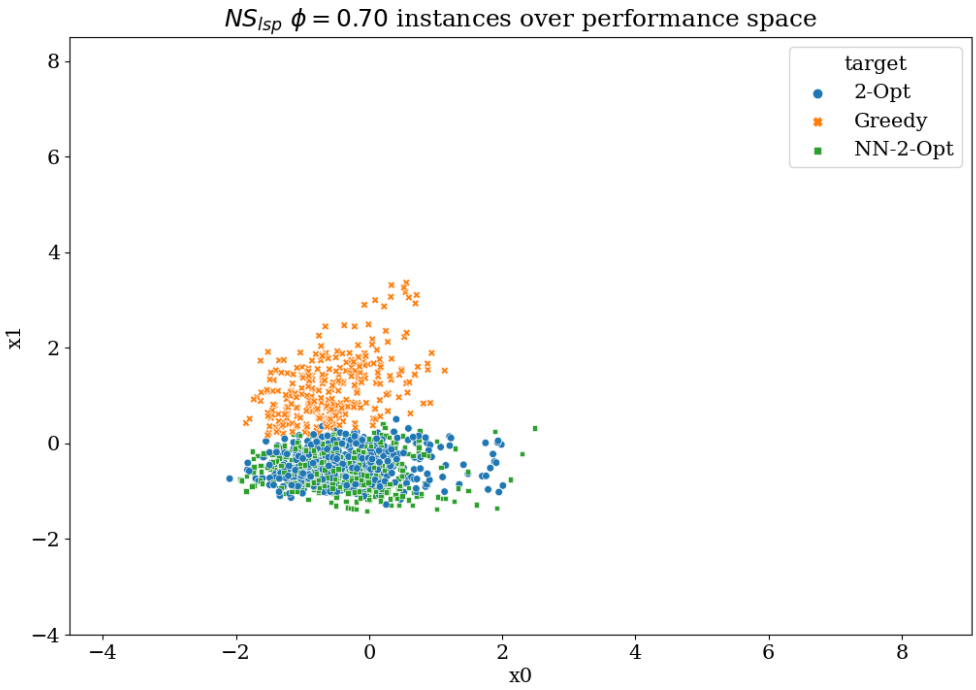


Figure B.7 Instance representation in a 2D space after applying PCAs to all the instances generated by NS_{lsp} setting $\phi = 0.70$. The colours and symbols reflect the 'winning' solver for each instance. Blue dots are the instances generated for 2-Opt, orange crosses Greedy, and green squares are the instances for NN-2-Opt.

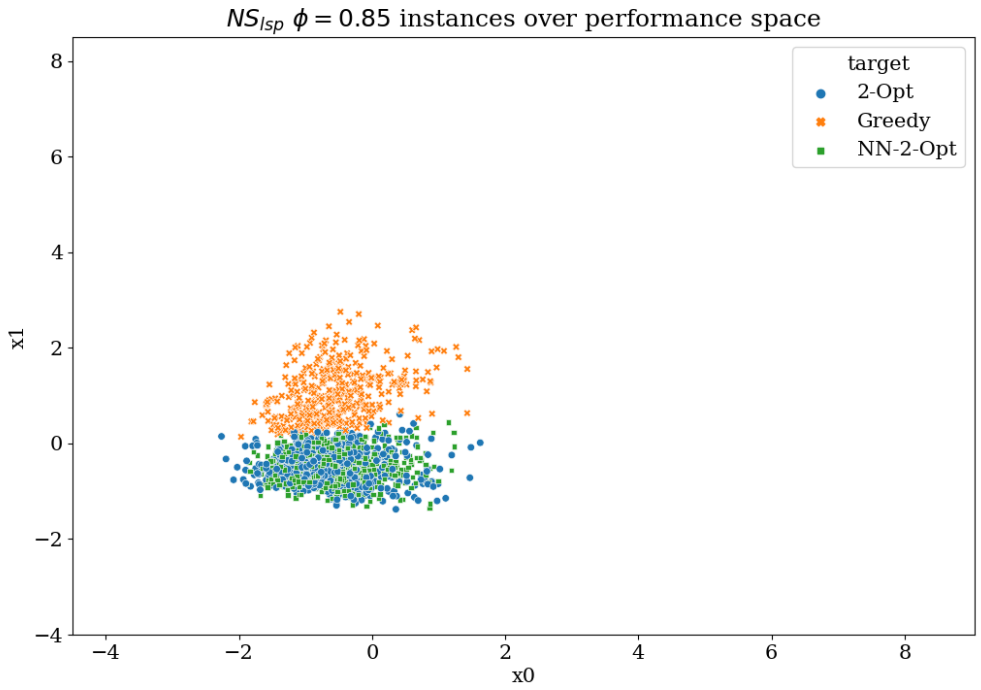


Figure B.8 Instance representation in a 2D space after applying PCAs to all the instances generated by NS_{lsp} setting $\phi = 0.85$. The colours and symbols reflect the 'winning' solver for each instance. Blue dots are the instances generated for 2-Opt, orange crosses Greedy, and green squares are the instances for NN-2-Opt.

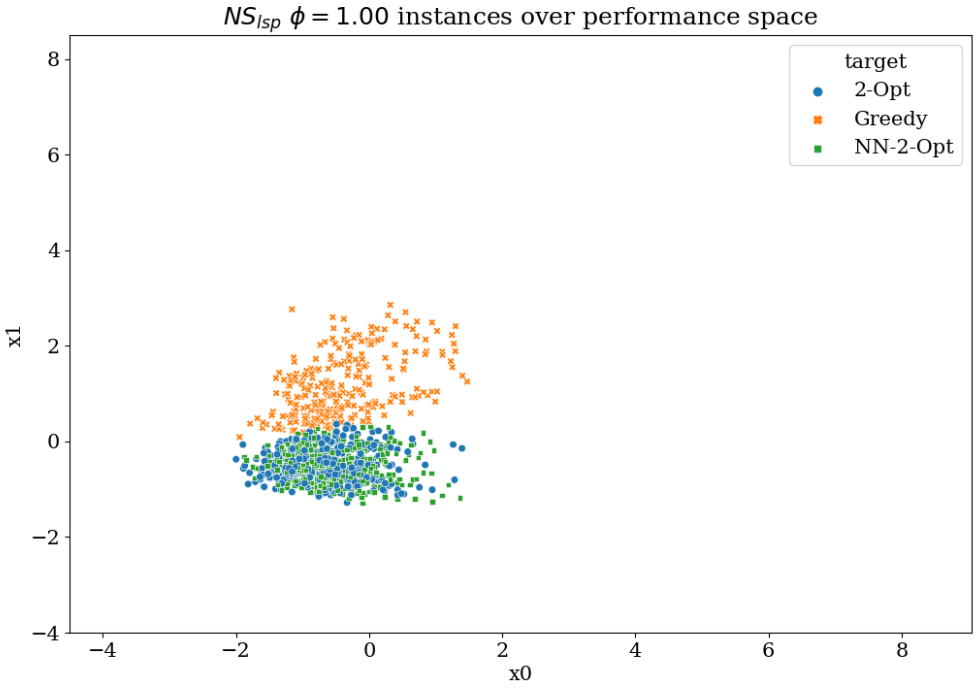
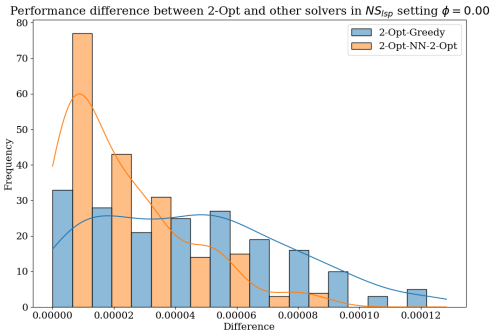
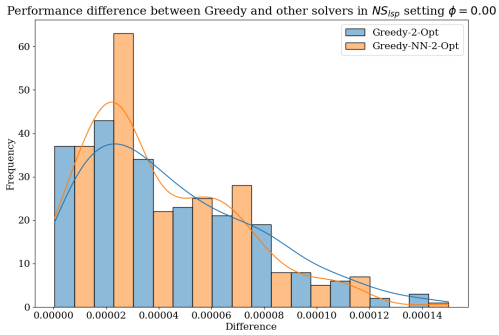


Figure B.9 Instance representation in a 2D space after applying PCAs to all the instances generated by NS_{lsp} setting $\phi = 1.00$. The colours and symbols reflect the 'winning' solver for each instance. Blue dots are the instances generated for 2-Opt, orange crosses Greedy, and green squares are the instances for NN-2-Opt.

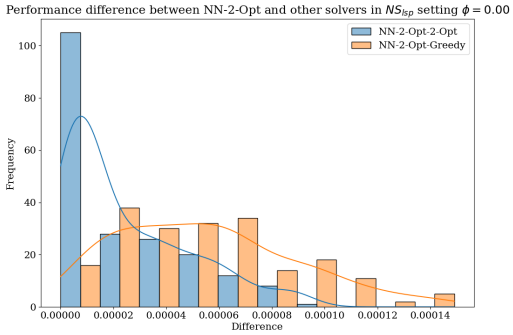
B.2 Performance-gap of NS_{lsp} Instances



(a) 2-Opt

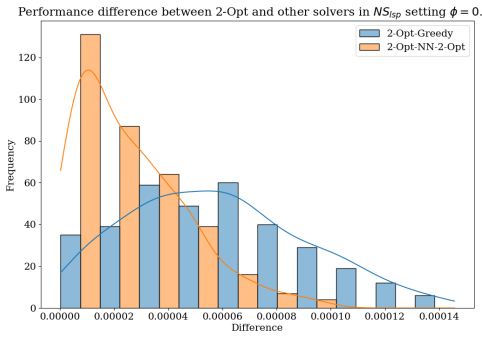


(b) Greedy

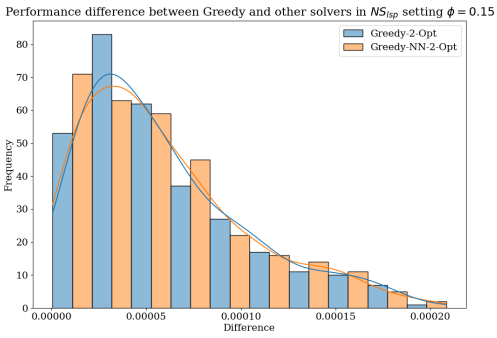


(c) NN-2-Opt

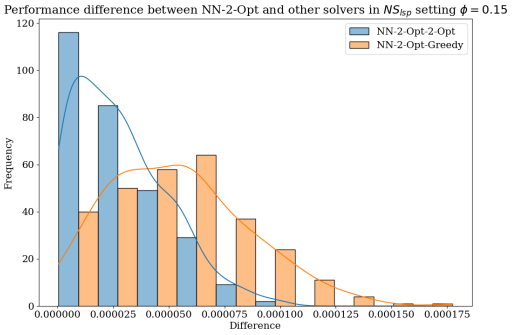
Figure B.10 Distribution of performance gap between the approaches 2-Opt, Greedy, and NN-2-Opt against other solvers in the portfolio by considering the instances generated for the former when running NS_{lsp} for the TSP domain when setting $\phi = 0.0$. The X-axis scale varies from one sub-figure to another to better display the bars in the plots. Therefore, the differences between algorithms depend on the solver that is taken as a reference.



(a) 2-Opt

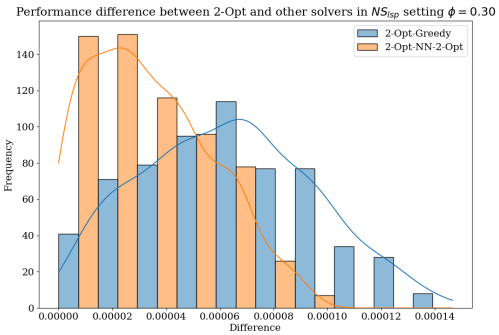


(b) Greedy

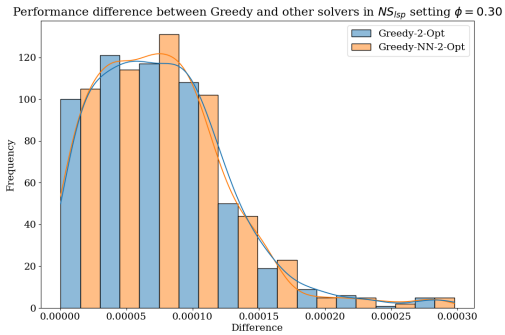


(c) NN-2-Opt

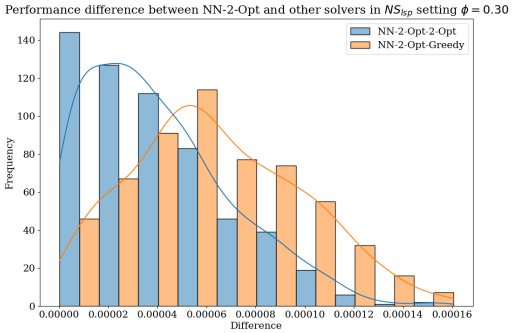
Figure B.11 Distribution of performance gap between the approaches 2-Opt, Greedy, and NN-2-Opt against other solvers in the portfolio by considering the instances generated for the former when running NS_{lsp} for the TSP domain when setting $\phi = 0.15$. The X-axis scale varies from one sub-figure to another to better display the bars in the plots. Therefore, the differences between algorithms depend on the solver that is taken as a reference.



(a) 2-Opt

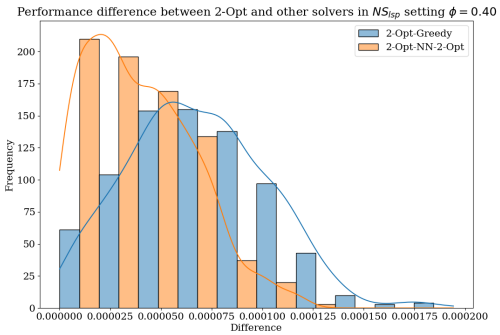


(b) Greedy

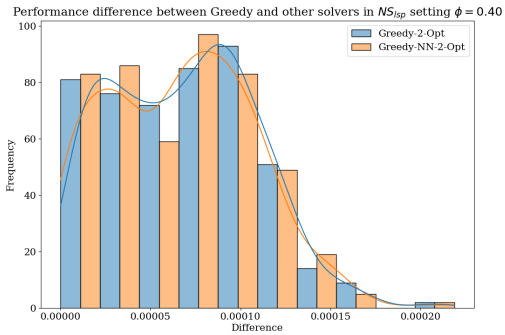


(c) NN-2-Opt

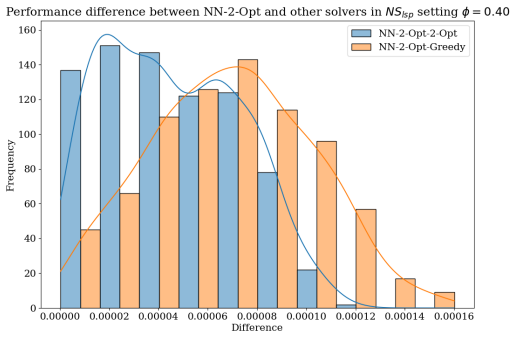
Figure B.12 Distribution of performance gap between the approaches 2-Opt, Greedy, and NN-2-Opt against other solvers in the portfolio by considering the instances generated for the former when running NS_{lsp} for the TSP domain when setting $\phi = 0.30$. The X-axis scale varies from one sub-figure to another to better display the bars in the plots. Therefore, the differences between algorithms depend on the solver that is taken as a reference.



(a) 2-Opt

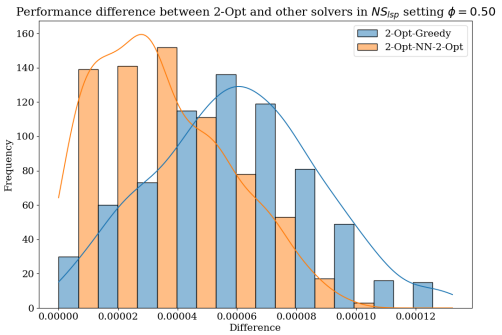


(b) Greedy

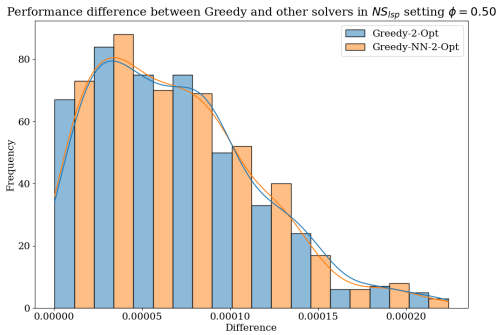


(c) NN-2-Opt

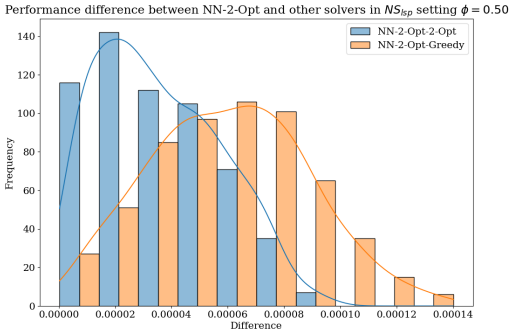
Figure B.13 Distribution of performance gap between the approaches 2-Opt, Greedy, and NN-2-Opt against other solvers in the portfolio by considering the instances generated for the former when running NS_{lsp} for the TSP domain when setting $\phi = 0.40$. The X-axis scale varies from one sub-figure to another to better display the bars in the plots. Therefore, the differences between algorithms depend on the solver that is taken as a reference.



(a) 2-Opt

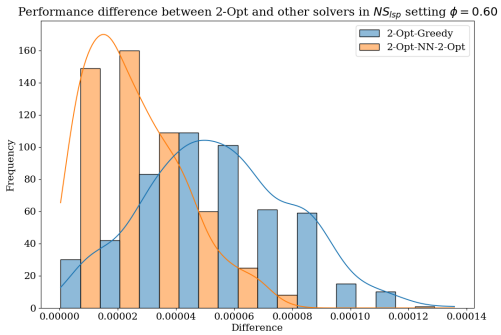


(b) Greedy

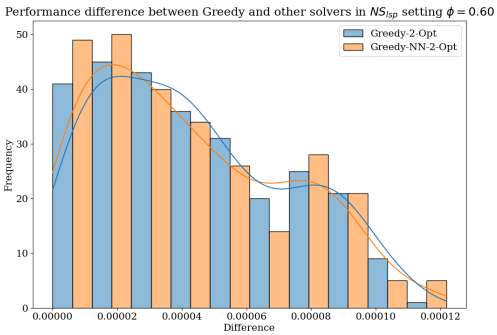


(c) NN-2-Opt

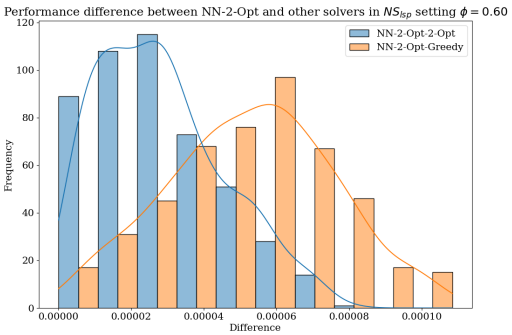
Figure B.14 Distribution of performance gap between the approaches 2-Opt, Greedy, and NN-2-Opt against other solvers in the portfolio by considering the instances generated for the former when running NS_{lsp} for the TSP domain when setting $\phi = 0.50$. The X-axis scale varies from one sub-figure to another to better display the bars in the plots. Therefore, the differences between algorithms depend on the solver that is taken as a reference.



(a) 2-Opt

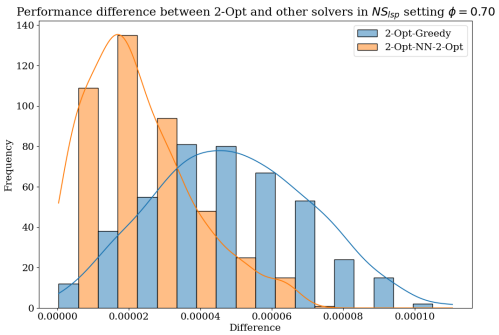


(b) Greedy

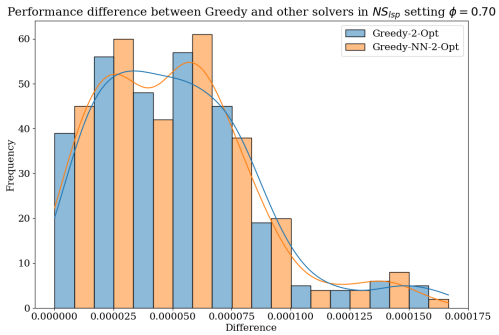


(c) NN-2-Opt

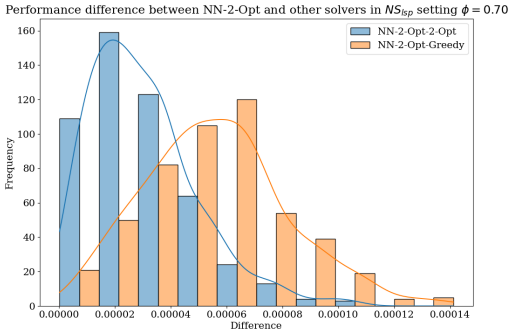
Figure B.15 Distribution of performance gap between the approaches 2-Opt, Greedy, and NN-2-Opt against other solvers in the portfolio by considering the instances generated for the former when running NS_{lsp} for the TSP domain when setting $\phi = 0.60$. The X-axis scale varies from one sub-figure to another to better display the bars in the plots. Therefore, the differences between algorithms depend on the solver that is taken as a reference.



(a) 2-Opt

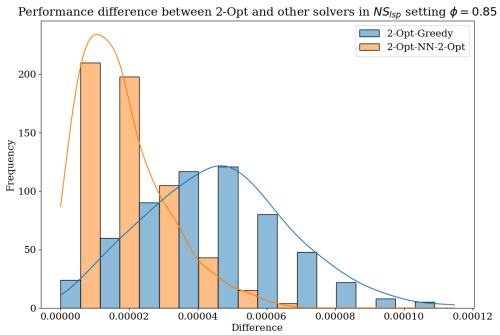


(b) Greedy

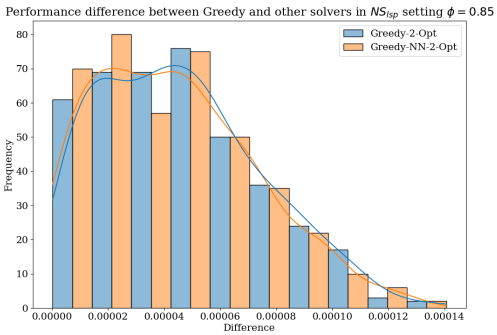


(c) NN-2-Opt

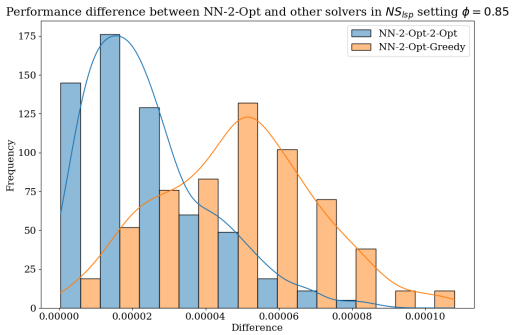
Figure B.16 Distribution of performance gap between the approaches 2-Opt, Greedy, and NN-2-Opt against other solvers in the portfolio by considering the instances generated for the former when running NS_{lsp} for the TSP domain when setting $\phi = 0.70$. The X-axis scale varies from one sub-figure to another to better display the bars in the plots. Therefore, the differences between algorithms depend on the solver that is taken as a reference.



(a) 2-Opt

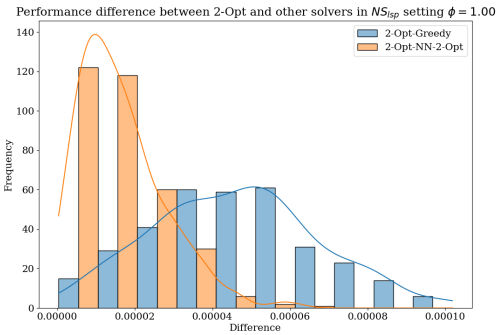


(b) Greedy

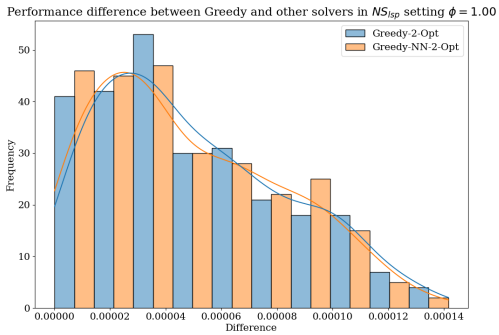


(c) NN-2-Opt

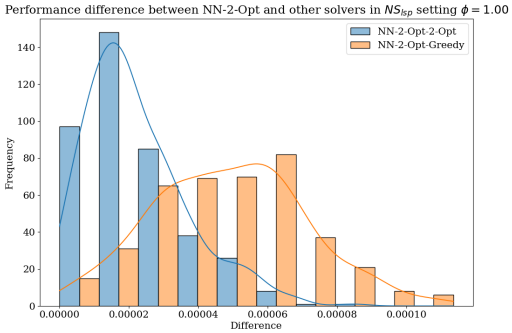
Figure B.17 Distribution of performance gap between the approaches 2-Opt, Greedy, and NN-2-Opt against other solvers in the portfolio by considering the instances generated for the former when running NS_{lsp} for the TSP domain when setting $\phi = 0.85$. The X-axis scale varies from one sub-figure to another to better display the bars in the plots. Therefore, the differences between algorithms depend on the solver that is taken as a reference.



(a) 2-Opt



(b) Greedy



(c) NN-2-Opt

Figure B.18 Distribution of performance gap between the approaches 2-Opt, Greedy, and NN-2-Opt against other solvers in the portfolio by considering the instances generated for the former when running NS_{lsp} for the TSP domain when setting $\phi = 1.00$. The X-axis scale varies from one sub-figure to another to better display the bars in the plots. Therefore, the differences between algorithms depend on the solver that is taken as a reference.