Dynamic Co-Evolutionary Algorithms for Dynamic, Constrained Optimisation Problems

by

Gary Pamparà

Dissertation presented for the degree of Doctor of Philosophy in the Faculty of Engineering at Stellenbosch University

Supervisor: Prof. A. P. Engelbrecht

March 2021

Declaration

By submitting this dissertation electronically, I declare that the entirety of the work contained therein is my own, original work, that I am the sole author thereof (save to the extent explicitly otherwise stated), that reproduction and publication thereof by Stellenbosch University will not infringe any third party rights and that I have not previously in its entirety or in part submitted it for obtaining any qualification.

Date: 1 March 2021

Abstract

Dynamic, constrained optimisation problems (DCOPs) are a class of optimisation problem where the problem landscape changes and problem constraints optionally change over time. Although DCOPs represent the super-set of optimisation problems, relatively little is understood about these problems due to the complexity added to the optimisation process. However, unlike the available optimisation algorithms developed for changing problem landscapes, few algorithm variants exist for DCOPs. Optimisation algorithms for DCOPs are expected to not only adapt to the changing problem landscape but need to also consider the feasibility of solutions whilst adapting to any changes to the problem constraints over time.

This thesis examines the constituent parts of the optimisation process by providing an in-depth review of the optimisation process. A substantial challenge is created by DCOPs for optimisation algorithms and this thesis examines these problems in detail with a fitness landscape analysis (FLA), before proposing a DCOP benchmark generator capable of producing a truly comprehensive set of possible problem landscape and constraint combinations. Quantification of the performance of optimisation algorithms on these comprehensive DCOP instances is identified as being problematic. The behaviour of the algorithm during the entire optimisation process should provide a better indication of algorithm performance and this thesis proposes a new measurement capable of quantifying algorithm performance from the very beginning of the optimisation process.

Generally, a constraint handling method is used to manage the optimisation problem constraints for the optimisation algorithm. The constraint handling method should also adapt to the changing optimisation problem. As a result, adaptive constraint handling methods are thought to be best. However, many of these methods introduce additional control parameters that require tuning which is not useful within DCOPs. This thesis provides evidence that a dynamic co-evolutionary approach using the Lagrangian transformation of the optimisation problem can produce solutions to DCOPs. The co-evolutionary approach uses dynamic optimisation algorithms in order to adapt to both the changing problem landscape and changing problem constraints. This thesis also proposes a novel self-adaptive quantum particle swarm optimisation as one of these dynamic optimisation algorithms.

Lastly, this thesis proposes a reproducible framework for computational intelligence allowing for the perfect replication of the experimental work of the aforementioned dynamic co-evolutionary algorithms.

Opsomming

Dinamiese, beperkte optimeringsprobleme (DCOPs) is 'n klas optimeringsprobleem waar die probleemlandskap verander en probleembeperkings opsioneel verander met die verloop van tyd. As gevolg van die addisionele kompleksiteit wat hierdie klas van optimeringsprobleem by die optimeringsproses byvoeg, word daar huidiglik relatief min verstaan omtrent dié superstel van optimeringsprobleme. Daar is 'n klein aantal optimeringsalgoritmes wat tans beskikbaar is vir veranderende probleemlandskappe soos DCOPs. Dit word van optimeringsalgoritmes vir DCOPs verwag om nie net by die veranderende probleemlandskap aan te pas nie, maar ook om die haalbaarheid van oplossings te oorweeg. Terselfdertyd moet optimeringsalgoritmes ook by enige verandering in die probleembeperkings aanpas met tyd.

Hierdie proefskrif ondersoek die dele van die optimeringsproses deur 'n diepgaande oorsig van die optimeringsproses te gee. Die uitdagings wat deur DCOPs aan optimeringsalgoritmes gestel word is groot, en hierdie proefskrif ondersoek hierdie probleme deur "fiksheidlandskapanalise" (FLA). Nadat die eienskappe van die probleme ondersoek is, word 'n funksie voorgestel wat DCOP probleme kan genereer. Hierdie funksie kan 'n reeks van moontlike probleemlandskap en beperkingskombinasies lewer. Die kwantifisering van optimeringsalgoritme oplossings wat vir die reeks probleme gevind word, kan as problematies geïdentifiseer word. Die versameling van resultate na elke iterasie van 'n algoritme gee 'n beter aanduiding van die kwalitiet van oplossings wat deur 'n algoritme gevind kan word. In hierdie proefskrif word 'n nuwe metingsproses voorgestel wat die kwalitiet van algoritme oplossings kan bepaal, vanaf die begin van die optimeringsproses tot en met die einde daarvan.

Oor die algemeen word 'n beperkingshanteringmetode gebruik om die probleem beperkings vir die optimeringsalgoritme te bestuur. Die beperkinghanteringsmetode moet ook aanpas by enige verandering van die optimeringsprobleem. As gevolg hiervan word aanpasbare metodes vir die hantering van probleembeperkings as die beste beskou. Ongelukkig benodig baie van die probleembeperking metodes addisionele beheerparameters wat ingestel moet word, maar die instel van beheerparameters maak geen sin binne DCOPs nie.

Hierdie proefskrif lewer bewys dat 'n dinamiese ko-evolusionêre benadering wat die Lagrangiaanse transformasie van die optimeringsprobleem gebruik, oplossings vir DCOPs kan lewer. Die ko-evolusionêre benadering gebruik di-

namiese optimeringsalgoritmes om aan te pas by die veranderende probleemlandskap en veranderende probleembeperkings. Hierdie proefskrif stel ook 'n nuwe selfaanpassende "kwantum deeltjieswerm optimeringsalgoritme" voor as een van hierdie dinamiese optimeringsalgoritmes.

Laastens stel hierdie proefskrif 'n raamwerk voor vir rekenaarintelligensie paradigmas wat die perfekte replikasie van die eksperimentele werk van die bogenoemde dinamiese ko-evolusionêre algoritmes moontlik maak.

Acknowledgements

I would like to express my sincere gratitude to the following people, without whom this would not have been possible:

- My supervisor, Professor Andries Engelbrecht, for always challenging me and providing crucial critique and guidance. Much of the motivation for this work is a result of his strong leadership and excellence. I am truly thankful for his support over the years.
- My family, for their love, support and trust. In particular my parents, Elvio and Suzette Pamparà, for their encouragement and support. To my brothers, Brett and Ian, and to my sister, Claire, for their compassion and understanding during this process. To Etienne and Charlene for their help whilst I was under pressure and to Stefan for the many discussions we had about our studies.
- To all my friends and colleagues whom encouraged me.
- All the CIlib developers and all research group members for the discussions, ideas and solutions to problems.
- My wife, Karen, for her unending love, support and patience; especially as we shared the highs and lows of our respective studies. For her always maintaining a sense of humour, together with our dogs (Penny and Donna) that kept me grounded.
- The Lord for providing me with everything that I need.

Contents

List of Figures

List of Tables

List of Algorithms

List of Listings

List of Scala REPL Sessions

Chapter 1

Introduction

The field of computational intelligence (CI) focuses on the study of methods to facilitate intelligent behaviour within optimisation problem search spaces. Within the field of CI, a multitude of methods exist which borrow inspiration from nature. Using such nature-inspired metaphors, search processes based on natural selection, the foraging behaviour of animals and the collective movement of collective groups of animals, amongst others, have been proposed. Although certain metaphors may be more popular and effective than others, attempts to create new metaphor based search processes continues apace. Although success has been achieved using these nature-inspired algorithms, no single algorithm is able to achieve success for all optimisation problem search spaces [304]. Optimisation problem complexity is determined by the problem landscape itself, the frequency of landscape change over time and by the restriction of feasible solutions by problem constraints that render portions of the search space infeasible. With the increasing complexity of optimisation problems the ability of an optimisation algorithm to find, maintain and ensure the feasibility of solutions becomes more challenging.

The remainder of this chapter is organised as follows. Section 1.1 provides further motivation for the investigation into the effect and influence of optimisation problem complexity on optimisation algorithm performance. The primary objectives of this thesis are listed in section 1.2, followed by the main contributions of the thesis in section 1.3. Finally, section 1.4 presents the detailed outline of this thesis.

1.1 Motivation

Real world problems introduce complexities that are often unique unto themselves. Complexities may include the size of the problem itself but are also often expressed as a set of constraints which limit the valid solutions that are possible for a particular problem search space. Problem constraints need not remain static and unchanging but can change over time. The changes may also

include the increase or decrease in the total number of problem constraints. The trading of shares on a stock market is probably the best example of a fluid system of constraints, where the availability, price and fees associated with stock market trading continuously change and adapt to the market itself. Furthermore, the trading process itself is variable with different strategies being more viable than others, albeit for a limited period of time. An increased understanding of the complex interactions between algorithm and problem search space is especially desirable, particularly when the problem search space is able to change whilst simultaneously being constrained. It is not known how the change in problem search space ultimately affects the optimisation algorithm, nor how the complexities of changing constraints possibly impede or encourage the search process of an optimisation algorithm.

In order to consider the complications presented by a dynamic optimisation problem with changing problem constraints, the research community attempts to model the problem characteristics using smaller, more understandable benchmark problems. Such benchmark problems are gradually extended to present more challenging scenarios for optimisation algorithms to provide solutions to. The extreme case for complexity is a totally dynamic optimisation problem that changes in multiple ways, possibly simultaneously. Such optimisation problems combine changing problem landscapes with changing problem constraints to produce the complex problem search spaces and are referred to as dynamic constrained optimisation problems (DCOPs). Moreover, optimisation algorithms may provide multiple possible solutions that differ in quality resulting in multi-modal problem search spaces [29, 84, 85, 101].

Optimisation algorithms have been successful in providing solutions to dynamic optimisation problems (DOPs) by adapting several optimisation algorithms originally designed to primarily consider static optimisation problems (SOPs), including genetic algorithms (GAs), differential evolution (DE) and particle swarm optimisation (PSO). Extensions to the canonical algorithm definitions have been proposed to allow the algorithms to operate within different problem domains, produce multiple solutions concurrently and to operate within changing problem landscapes. Although the success of the modified algorithms has been mixed, these algorithm variants are nonetheless viable for changing problem landscapes. DCOP landscapes provide an increase in optimisation problem complexity that is highly challenging for existing dynamic algorithm variants. Furthermore, due to the existence of infeasible problem search space regions, a constraint handling approach is required to allow for the effective search of an optimisation algorithm. The final result is an interaction between three different parts of the optimisation process and the influence of each of these aspects is not yet understood. Moreover, the existing benchmark problems are not of a sufficient complexity to allow for the study, development and improvement of optimisation algorithms for this classification of problem.

When considering the optimisation process as a whole, interactions between different parts of the process should remain fixed whilst allowing another to

be variable in order to attempt to discern possible patterns. The empirical results should therefore present only the effects of the optimisation algorithm on the optimisation process. By limiting the variability within the empirical work, the study how the algorithms are influenced may be conducted without allowing other unwanted side-effects. The algorithmic interaction will include the effects of constraint handling approaches to determine the challenge of the DCOP benchmark problem.

1.2 Objectives

The primary objectives of this thesis are summarised as follows:

- Provide a thorough overview of complexity within the optimisation process by considering the optimisation problem, optimisation algorithm and constraint handling approach for problem landscapes from simple to the most complex.
- Investigate the currently available benchmark problems for changing problem landscapes, together with the inclusion of problem constraints, proposing improvements where required.
- Provide fitness landscape analysis results for benchmark DCOP instances.
- Determine the effectiveness of performance measures for dynamically changing problem search spaces, considering the impact of problem level constraints in order to evaluate algorithm performance in complex landscapes.
- Examine the effectiveness of optimisation algorithms together with constraint handling approaches in order to provide solutions to a comprehensive set of dynamic and constrained benchmark problems.
- To develop a system that allows for the correct and reproducible investigation of the optimisation complexity present within dynamic, constrained optimisation problems.

1.3 Contributions

The main contributions of this research are as follows:

1. A benchmark problem generator function, known as the constrained moving peaks benchmark (CMPB) generator, which is capable of generating 784 unique DCOP instances.

2. A new performance measure, P_{RED}, which provides a holistic way of comparing algorithm performance for DOPs and for DCOPs. The measure produces a vector, referred to as the *performance profile*, allowing for the comparison of optimisation algorithms in changing problem landscapes.

3. A variant of PSO for DOP, known as self-adaptive quantum particle swarm optimisation (SaQPSO) which self-adapts the size of the quantum cloud based on the current algorithm performance.

4. A dynamic co-evolutionary framework from which optimisation algorithms for DCOPs can be derived. From this dynamic co-evolutionary framework a total of six new algorithms are proposed to provide solution to DCOP.

5. A software library, known as CIlib, which allows for the exact replication of experimental results. Through CIlib the complexities of DCOPs, as well as the interactions of optimisation algorithms on the optimisation problems are controlled in a rigorous manner, preventing possible errors in the experimental work.

1.4 Thesis Outline

The thesis is split into separate parts that address different concerns for the thesis as a whole and is organised as follows:

- **Part I**: Background
  - **Chapter 2** provides background information about the optimisation process and highlights the inter-dependence of the problem, constraints and algorithm.
  - **Chapter 3** provides a review of optimisation algorithms, highlighting the progression of algorithms to cater for changing problem search spaces and the handling of constraints.
- **Part II**: Complexity of Dynamic, Constrained Optimisation Problems
  - **Chapter 4** provides background information about currently available benchmark problems and generators for dynamic, constrained optimisation problems together with their shortcomings.
  - **Chapter 5** proposes a new DCOP benchmark problem generator which provides a comprehensive representation of both the changing optimisation problem landscape and the changing optimisation problem constraints.

  - **Chapter 6** presents an approach to allow for a vector-based comparisons between algorithms. This vector-based comparison method allows for a more holistic representation of the actual performance of an optimisation algorithm for a changing optimisation problem with optional changing problem constraints.

- **Part III**: Algorithm Performance on Dynamic, Constrainted Optimisation Problems

  - **Chapter 7** proposes a variant of the quantum particle swarm optimisation (QPSO) for dynamic optimisation problems which self-adapts an algorithm control parameter based on information obtained from the candidate solutions about the optimisation problem landscape.
  - **Chapter 8** discusses the use of co-evolution to solve static, constrained optimisation problems. This chapter continues by proposing an adjustment to the co-evolutionary algorithm formulation to provide solutions to dynamic, constrained optimisation algorithms.
  - **Chapter 9** discusses the evaluation and statistical procedure for the empirical work within this thesis. The chapter discusses the comparison process of optimisation algorithms on a comprehensive set of DCOPs, use of vector-based performance measures as well as the statistical comparison procedure that was followed.
  - **Chapter 10** presents an empirical investigation into the performance characteristics of optimisation algorithms attempting to solve changing problems spaces that are also constrained. This chapter builds upon previous chapters.

- **Part IV**: Reproducible Computational Intelligence

  - **Chapter 11** presents the rationale for a research methodology which places emphasis on reproducible results and compares the current state-of-the-art software for CI and machine learning. The work presented in this chapter defines the foundation from which the empirical work, presented in chapter 10, is based.
  - **Chapter 12** presents a new approach to CI software that is designed with reproduction and repeatability in mind, resulting in a monadic software library with a focus on composition. This chapter discusses how the goals and principles from chapter 11 have materialised by presenting the logical structure of, and examples for, the CIlib software library.

- **Chapter 13** provides concluding remarks and presents avenues for future work.

The following appendices are also provided:

- **Appendix A** lists the algorithm ranking results from the empirical analysis of algorithm performance on dynamic, constrained optimisation problems.
- **Appendix B** provides the listing of all acronyms.
- **Appendix C** lists the derived publications from this thesis.

Part I

Background

Chapter 2

Optimisation

> *One of the most important characteristics of complex adaptive systems is that they cannot, in general, be successfully analysed by determining in advance a set of properties or aspects that are studied separately and then combining those partial approaches in an attempt to form a picture of the whole. Instead, it is necessary to look at the whole system, even if that means taking a crude look, and then allowing possible simplifications to emerge from the work.*
>
> Murray Gell-Mann, Santa Fé Institute

Optimisation defines an action which makes the most effective use of a situation or resource. Although this definition is vague, it is the general description of optimisation without any context. Computational optimisation techniques refine the optimisation definition to selecting the best element (based on some criterion) from a set of available alternatives. Within the context of stochastic optimisation, the three participants in the optimisation process include the optimisation problem itself, potential constraints applied to the optimisation problem, and the optimisation algorithm. An inter-dependence exists between the main parts of the optimisation process as illustrated in figure 2.1. Due to this inter-dependence no single part of the optimisation process can be considered in complete isolation, but in the sections that follow each main aspect of the optimisation process will be discussed.

Section 2.1 discusses optimisation problem definitions that range from static optimisation problems without constraints to dynamic optimisation problems with changing problem constraints. The categorisation of dynamic optimisation problems is discussed in section 2.2, with constrained optimisation problems discussed in section 2.3. Section 2.4 concludes the chapter.

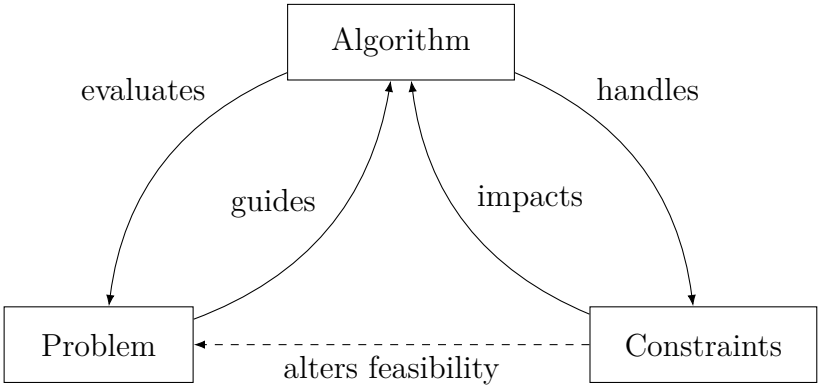


Figure 2.1: Interdependence of the components of the optimisation process

2.1 Optimisation Problems

The purpose of the process of optimisation is to locate a solution to the specific problem that is as optimal as possible. An optimisation problem consists of an objective function which is used to determine the quality of a solution and a set of problem constraints which determine if the solution may be considered. Optimisation algorithms attempt to locate and maintain solutions to the optimisation problem.

Assuming minimisation, a SOP is formally defined as:

Definition 2.1 Boundary-constrained Static Optimisation Problem**.**

$$\begin{aligned} &\text{minimize} && f(\mathbf{x}) \\ &\text{subject to} && \mathbf{x} = (x_1, x_2, \ldots, x_n), \\ & && x_i \in [x_{i,min}, x_{i,max}], \quad i = 1, 2, \ldots, n \end{aligned} \tag{2.1}$$

where the objective function is

$$f : \mathcal{S} \rightarrow \mathbb{R}^n \tag{2.2}$$

The vector $\mathbf{x}$ specifies the problem decision variables, x_i, for problem dimensions $i = 1, 2, \ldots, n$. The objective function, f, quantifies the quality of the decision variables within the search space $\mathcal{S}$. Each vector of decision variables represents a *candidate solution* to the static optimisation problem. The purpose of the optimisation process is to find a global minimum solution, $\mathbf{x}^* \in \mathcal{S}$, such that for all $\mathbf{y} \in \mathcal{S}$:

$$f(\mathbf{x}^*) \leq f(\mathbf{y}) \tag{2.3}$$

Minimisation and maximisation optimisation problems may be interchanged provided the following relationship is enforced:

$$\text{maximise } f(\mathbf{x}) \equiv \text{minimise } (-f(\mathbf{x})) \tag{2.4}$$

Constraints define which values within the domain of decision variables will result in a feasible solution. The constraints effectively divide the search space, $\mathcal{S}$, into feasible and infeasible regions, respectively denoted as $\mathcal{F}$ and $\mathcal{I}$, such that $\mathcal{S} = \mathcal{F} \cup \mathcal{I}$. Such problems are referred to as constrained optimisation problems (COPs) and are defined as:

Definition 2.2. *Constrained Optimisation Problem*

$$\begin{aligned} \text{minimize} \quad & f(\mathbf{x}) \\ \text{subject to} \quad & g_m(\mathbf{x}) \leq 0, && m = 1, 2, \dots, n_g, \\ & h_m(\mathbf{x}) = 0, && m = n_g + 1, n_g + 2, \dots, n_g + n_h, \\ & \mathbf{x} = (x_1, x_2, \dots, x_n), \\ & x_i \in [x_{i,min}, x_{i,max}], && i = 1, 2, \dots, n \end{aligned} \tag{2.5}$$

where n_g and n_h are the number of inequality and equality constraints.

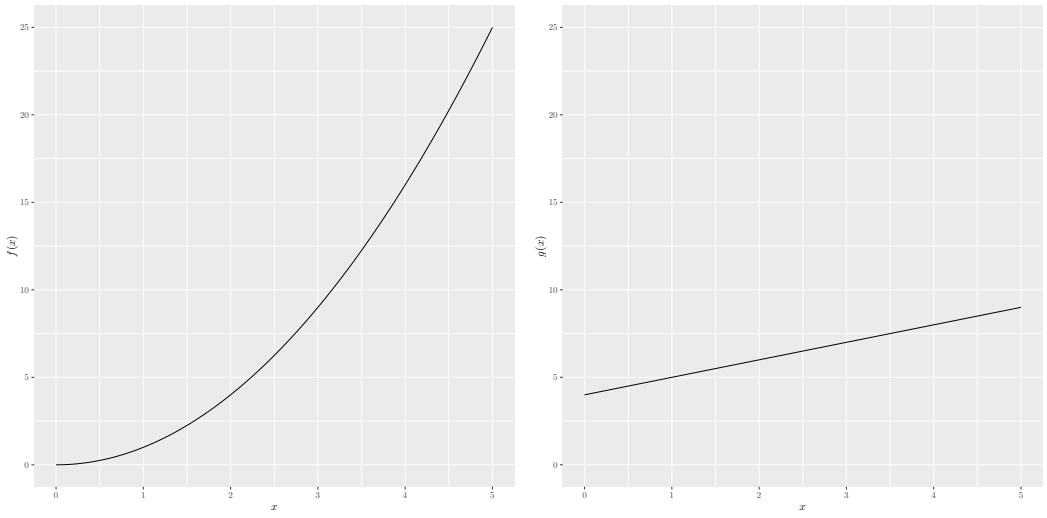
Candidate solutions within the search space that are located within infeasible regions of the problem search space are referred to as infeasible solutions. Infeasible solutions violate at least one of the defined inequality or equality constraints of the COP, irrespective of objective function value. When searching for solutions to a COP, the goal of the optimisation process is to find solutions that achieve the best trade-off between solution quality (determined by $f(\mathbf{x})$) and constraint violation.

Application of constraints to an optimisation problem produces a dual perspective of the problem search space. Evaluation of the objective function of a COP results in the *objective landscape* of the problem space. However, by evaluating a function that quantifies the degree of constraint violation allows for the observation of the optimisation problem's *constraint landscape* [180]. Each perspective of the problem landscape can therefore be considered in isolation. Figure 2.2 illustrates a COP from the perspective of both the objective and constraint landscapes, where $f(x) = x^2$ and $g(x) = x + 4 \leq 0$.

Static optimisation problems maintain a constant, unchanging objective function. In contrast, a DOP represents an objective function which changes over time. The problem is formally defined (assuming minimisation) as:

Definition 2.3. *Boundary-constrained Dynamic Optimisation Problem*

$$\begin{aligned} \text{minimize} \quad & f(\mathbf{x}, \varpi(t)) \\ \text{subject to} \quad & \varpi(t) = (\varpi_1, \varpi_2, \dots, \varpi_n), \\ & \mathbf{x} = (x_1, x_2, \dots, x_n), \\ & x_i \in [x_{i,min}, x_{i,max}], \quad i = 1, 2, \dots, n \end{aligned} \tag{2.6}$$



(a) Objective landscape **(b)** Constraint landscape

Figure 2.2: Search space perspectives of a constrained optimisation problem

where $\varpi(t)$ is a vector of time-dependent objective function parameters. The goal is to find

$$\mathbf{x}^*(t) = \min f(\mathbf{x}, \varpi(t)) \tag{2.7}$$

where $\mathbf{x}^*(t)$ is the optimum found at time-step t, whilst continuing to track the optimum's trajectory.

Static optimisation problem search spaces are a special case of dynamic optimisation problem search spaces, where t remains constant and does not change. The *period* between dynamic optimisation problem search space changes (known as the *change period*) may be treated as an instance of a static optimisation problem search space. As the problem search space changes over time, the manner in which the DOP changes increases the difficulty in tracking how optima change within the problem search space. The following characteristics all increase the complexity of maintaining a solution within a DOP:

- **Optima change direction**: After the optimisation problem experiences a change, optima within the problem search space may be different from those before the change. Optima may have moved to different locations, the value of optima could change, or a combination of these effects have occurred. Such changes to optima increase the complexity of the problem. When optima change, the current best solution may no longer be relevant, whereas if optima relocate, the current solution is potentially not near optima.

- **Change pattern**: A pattern may exist within a sequence of problem search space changes. Repeating patterns allow for the calculation of a *cycle length* [23] between pattern repetitions within the problem search space. Observed patterns provide additional information to the optimisation process which may be exploited to aid in finding solutions. If a pattern is known in advance, the optimisation process is able to exploit that knowledge to find solutions, predicting the changes the repeating pattern introduces. Upfront knowledge of change patterns within the problem search space is, however, not guaranteed and these unknown patterns instead increase the complexity of the optimisation problem. When the optimisation algorithm is able to track the pattern of changing optima within the search space, the impact of this problem space complication is minimised.

- **Change coherence**: Optimisation problem changes may adjust the optima of the problem in the same or in different ways. In a *homogeneous* problem search space all optima have the same change transformation applied. If the optimisation process can identify that optima have changed in the same way it may be beneficial to process, but the same change experienced may be applied differently to different optima. A *heterogeneous* problem search space allows optima to have different change transformations. Again, such changes increase the problem complexity by making solutions more difficult to find.

- **Temporal severity**: Changes to the optimisation problem search space occur at a specific frequency. Optimisation problem changes having a high temporal severity indicate frequent search space changes and shorter change periods. Low temporal severity describes search spaces that have larger change periods and remain static for longer periods of time. Shorter change periods prevent the optimisation process from locating better solutions before the next change, whereas longer change periods allow more time for the optimisation process to locate solutions. As a result, the complexity of the optimisation problem is increased by varying the frequency of changes.

- **Spatial severity**: The magnitude of optima change within a dynamic optimisation problem can vary from subtle to extreme. Extreme changes within the optimisation problem alter the problem search space in a severe manner whereby the search spaces before and after a change may be unrecognisable to the optimisation process. Subtle changes produce less severe alterations to the problem search space which make it simpler for the optimisation process to adjust to. The severity of spatial changes also increases the complexity of the optimisation problem.

The change permutations that a dynamic optimisation problem could experience due the an environment change can be considered limitless. Although DOPs change over time, the optimisation problems may still have constraints that divide the problem search space into feasible and infeasible regions. Combining optimisation problem constraints with DOPs results in DCOPs:

Definition 2.4. *Dynamic Constrained Optimisation Problem*

$$\begin{aligned} \text{minimize} \quad & f(\mathbf{x}, \varpi(t)) \\ \text{subject to} \quad & \varpi(t) = (\varpi_1, \varpi_2, \dots, \varpi_n), \\ & g_m(\mathbf{x}, \varpi(t)) \leq 0, \qquad m = 1, 2, \dots, n_g, \\ & h_m(\mathbf{x}, \varpi(t)) = 0, \qquad m = n_g + 1, n_g + 2, \dots, n_g + n_h, \\ & \mathbf{x} = (x_1, x_2, \dots, x_n), \\ & x_i \in [x_{min}, x_{max}], \quad i = 1, 2, \dots, n \end{aligned} \tag{2.8}$$

where $\varpi(t)$ is a vector of time-dependent objective function parameters. The goal is to find

$$\mathbf{x}^*(t) = \min f(\mathbf{x}, \varpi(t)) \tag{2.9}$$

where $\mathbf{x}^*(t)$ is the optimum found at time-step t, whilst continuing to track the optimum's trajectory.

Because both the optimisation problem and constraints may independently change over time, the feasible and infeasible regions of the search also change. These "pockets" of feasible and infeasible space, respectively denoted as $\mathcal{F}_r$ and $\mathcal{I}_r$, may change from feasible to infeasible, or vice versa. Moreover, search space locations that are currently infeasible may become feasible, potentially revealing the new global optimum solution when the search space experiences a change. The number of feasible and infeasible pockets, $n_\mathcal{F} = |\mathcal{F}|$ and $n_\mathcal{I} = |\mathcal{I}|$, differ over time whilst maintaining the following search space invariants:

$$\mathcal{F} = \bigcup_{r=1}^{n_\mathcal{F}} \mathcal{F}_r \qquad\qquad \mathcal{I} = \bigcup_{r=1}^{n_\mathcal{I}} \mathcal{I}_r$$

such that $\mathcal{S} = \mathcal{F} \cup \mathcal{I}$.

The definition of DCOPs allows for the optimisation problem constraints to optionally change over time in addition to the optimisation problem search space. If the constraints within a DCOP also change over time, the complexity of the optimisation problem increases beyond the complexity introduced by changing the problem search space over time. DCOPs allow for four different optimisation problem combinations from the least amount of complexity to the most complexity:

- Static objective function with static constraints (SOSC)

- Static objective function with dynamic constraints (SODC)
- Dynamic objective function with static constraints (DOSC)
- Dynamic objective function with dynamic constraints (DODC)

The objective function and constraint combinations for DCOPs are illustrated in figure 2.3.

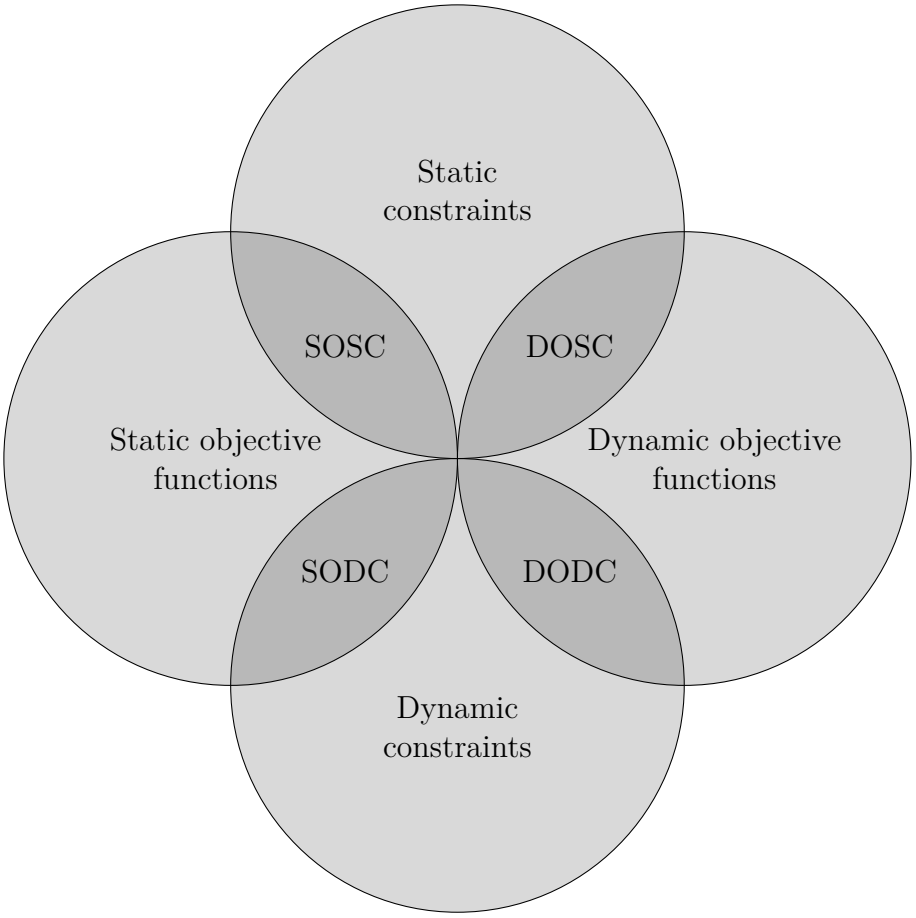


Figure 2.3: Classes of constrained optimisation problems

2.2 Dynamic Optimisation Problem Classifications

To understand the characteristics of DOPs, different categorisation schemes have been proposed [3, 77, 81, 123]. Sections 2.2.1 to 2.2.5 discuss these classification schemes.

2.2.1 Angeline's Classification

Angeline [3] classified DOPs based on the observed patterns of problem search space change. The classification categorises the trajectory of optima as they move through the problem search space. The following optima trajectory types are considered:

- **Linear** trajectories describe optima that move in a straight line from an initial location within the search space. Movement step sizes are calculated relative to the initial optima location, with the step size increasing for each change experienced by the optimisation problem.
- **Circular** trajectories move optima in a circular, orbit-like shape. The movement of the optima defines a periodic shift in the optima location that returns to the initial starting point. A repeating sequence of random movements that terminate at the same starting location is also defined as circular [297].
- **Random** optima trajectories display no discernible pattern. The optima move freely throughout the problem search space and display no movement pattern.

The classification scheme by Angeline [3] focuses on optima location and optima movement across search space change periods. Although the classification does describe how optima move over time within the search space, it neglects the possibility of value changes in optima.

2.2.2 Classification of Hu & Eberhart and Shi & Eberhart

Eberhart and Shi [81] and Hu and Eberhart [123] derived a classification scheme based on how optima are modified when the search space experiences change. The following modifications were identified:

- **Type I** search spaces have optima which change their location within the search space. Movement of optima maintains the current optima value, whilst relocating the optima to a different position within the problem search space. The number of optima within the problem search space remains the same.
- **Type II** search spaces maintain the location of optima but allow for the value of optima to change. The global optimum is the optimum with the best objective function value, and therefore may switch to a different optimum location as a result to the search space change. As with type I search spaces the number of optima within the problem search space does not change with this change in the environment. It is, however,

possible that the change in optima values simulate the disappearance of an optimum. For example, within a search space of multiple peaks, a search space change may flatten a peak completely thereby simulating the disappearance of an optimum.

- **Type III** search spaces combine the optima movements of both type I and type II search spaces. Both optima locations and values may be modified as the search space experiences change.

This classification scheme is able to classify optima movement and value change within a DOP. Heterogeneous optima are also possible using this classification, without considering the severity of spatial changes.

2.2.3 Weicker's Classification

Weicker [297, 298] proposed a classification framework for DOPs which identifies the following features:

- The nature of the optimisation problem as being either static or dynamic.
- Spatial severity of the optimisation problem is either constant or variable.
- The periodic nature of the optimisation problem; whether a search space returns to a common set of previous states.
- Whether the global optimum of the optimisation problem alternates between different optima as the search space changes.
- Whether optima modifications are homogeneous or heterogeneous upon search space change.

Weicker's classification framework provides more general information about the dynamic optimisation problem search space than either classification of Angeline or Eberhart *et al.* The classification framework does not, however, subsume all criteria defined by Angeline and Eberhart *et al.*, thereby necessitating the continued use of the previous classification schemes.

2.2.4 De Jong's Classification

De Jong [62] defined a nature inspired metaphor to describe the changes within dynamic optimisation problem search spaces. These search space descriptions include:

- Drifting landscapes which change gradually over time. Changes are small and highly frequent, allowing for simple tracking by an optimisation algorithm.

- Drastic morphological changes describe optimisation problem search spaces which experience significant change. Problem search space areas previously deemed uninteresting may contain new optima after a change. Such problem search spaces present a high severity of change.
- Landscapes which present cyclical patterns, repeating previously observed problem search spaces.
- Fundamental landscape change which is abrupt and potentially discontinuous. Change severity is larger than that of landscapes with significant morphological change.

De Jong's classification provides opaque descriptions for problem search spaces. The discernible difference between the mentioned spatial severities of the classification is small and requires further disambiguation. The classification focuses on the severity of landscape change stating that the changes are significant or fundamental landscape changes; without specifying if changes are cumulative or immediate. Although De Jong's classification could be used to describe the search space changes, it is not granular enough. Problem search space changes that conform to multiple categories outlined by De Jong may have very different environment changes.

2.2.5 Classification of Duhain and Engelbrecht

Duhain and Engelbrecht [77] proposed a more comprehensive classification scheme. The classification scheme combines the classifications of Angeline [3], Eberhart and Shi [81], and Hu and Eberhart [123], but also includes the severity of both temporal and spatial changes. The resulting classification broadly divides problem search spaces into four base environment types based on spatial and temporal changes (illustrated in figure 2.4):

- **Quasi-static** problem search spaces change with low spatial and low temporal severities. A problem search space is *static* if either spatial or temporal severity has a value of zero. When one severity is zero it effectively disables the effect of the other severity in applying a change to the problem search space. For example, problem search spaces with a non-zero temporal severity and zeroed spatial severity continue to experience temporal changes but produce the same problem search space after applying an environment change.
- **Progressive** problem search spaces frequently change and have a high temporal severity. The spatial changes are small producing optima within the problem search space that gradually move over time within the problem search space bounds.

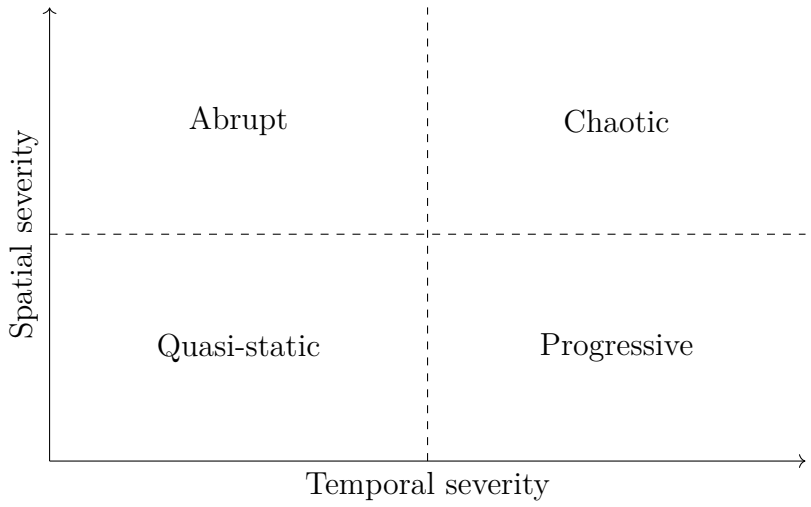


Figure 2.4: Severity classification types

- **Abrupt** changes to a problem search space create infrequent but large changes. In other words, the problem search space remains static for a period of time before drastically changing.
- **Chaotic** problem search spaces observe large changes to the problem search space frequently.

The combination of classification schemes together with the spatial and temporal severity classification produces 27 unique problem environment instances. These problem search space instances describe the movement of optima, patterns in subsequent problem search spaces and the change severities of a problem search space. Duhain and Engelbrecht [77] generated the unique problem instances using a base landscape function defined by the moving peaks benchmark (MPB) function generator [18]. A succinct problem search space description is also possible as a by-product of the classification scheme. Describing a problem search space as "chaotic, type I, circular" specifies that the problem search space contains a circular trajectory pattern for optima, with unchanging optima values, within a problem search space that changes with a chaotic frequency. The description may become more succinct through the use of a three letter acronym. The acronym for the "**C**haotic, type **I**, **C**ircular" problem search space previously mentioned is simply `C1C`.

2.3 Constrained Optimisation

The management of constraints within the optimisation problem depends on the type of problem constraint, the approach used to handle the problem constraints and how constraints impact the computational effort of the optimisation algorithm. The sections that follow discuss each of these considerations.

2.3.1 Types of Constraints

Different types of constraints can be applied to optimisation problems:

- **Boundary constraints** define the upper and lower bound values for each decision variable of the optimisation problem. The boundaries for each dimension enforce a hyper-cube within which possible problem solutions must be contained.
- **Inequality constraints** specify relations that determine the feasibility of optimisation problem decision variables.
- **Equality constraints** specify relations for which the problem decision variables must be equal to. An equality constraint can be transformed into an inequality constraint through the transformation

$$|h_m(\mathbf{x})| - \epsilon \leq 0 \tag{2.10}$$

  where $\epsilon > 0$ is a small tolerance value.

Although definition 2.4 indicates that equality and inequality constraints are functions that accept the complete solution vector, the entire input vector need not be used within the constraint function. For example, a large portion of the decision variables could be ignored to explicitly state a dependency between just two decision variables.

2.3.2 Constraint Handling Approaches

As previously mentioned, constraints restrict the possible set of feasible solutions within a problem search space. Any optimisation algorithm should guide the search to parts of the problem search space that balance the trade-off between solution quality and constraint violation. Ideally, no problem constraints should be violated within a solution but in situations where only infeasible solutions are possible, solutions that least violate the problem constraints are preferred. A board categorisation of constraint handling approaches is given in table 2.1.

Table 2.1: Constraint Handling Approaches

Classification	Discussion & Critique
Penalty based regularisation approaches	Regularisation approaches change the constrained optimisation problem into a boundary constrained optimisation problem by adding a penalty function to the objective function. Multiple different approaches to penalty functions exist which include static [7, 119, 122, 152, 219, 243] and dynamic [33, 100, 131, 142, 192, 257] penalty functions to adaptive [48, 107, 262] and self-adaptive [45, 88, 90, 227, 287] penalty functions. Penalty values should always maintain the *minimum penalty rule* [61, 260] where the penalty is kept as low as possible, just above the limit below which solutions are infeasible. Alternative penalty strategies also exist whereby the penalty and objective functions are balanced [66, 249] or where infeasible solutions are simply replaced [48, 66, 191]. Penalty based approaches have the distinct advantage of being simple to implement, as well as to apply to optimisation problems [46]. Unfortunately, most approaches are problem dependent, making the design of good penalty functions challenging [257].
Algorithmic regularisation approaches	Regularisation approaches that convert the constrained optimisation problem into a boundary-constrained optimisation problem by reformulating the problem into a different class of problem. In such scenarios, the optimisation algorithm is directly responsible for the management of problem level constraints instead of merely satisfying them. It is commonplace that the constraints become objective functions for the optimisation algorithm to consider in addition to the original objective function. Multi-objective optimisation (MOO) [71, 84, 89] and co-evolutionary approaches [125, 227] are examples where the optimisation problem has been reformed to suit the optimisation algorithm.

Table 2.1: (continued)

Classification	Discussion & Critique
	Reformulating the optimisation problem as multiple different objective functions, considered cooperatively or competitively, is a more complex approach to obtaining a solution to the problem. With the increased algorithm complexity, in addition to the complexity introduced by the COP, such reformulation approaches are not always appropriate. An advantage to such a formulation is that multiple solutions may be found, presenting a Pareto-front which represents the trade-offs between the objective and constraint functions.
Domain specific representations	It may be preferable to represent a solution to an optimisation problem using a representation that more accurately describes the domain. Changing representation may also allow for the constraints to be directly catered for by algorithm operators that are also aware of the domain [100, 145, 204]. Examples include problem-specific operators such as those used in the constraint consistency approach [151], decoders [60] and homomorphous mappings [146, 198]. Representational changes have the benefit of more accurately describing the domain of the problem, whilst at the same time potentially preventing the creation of infeasible solutions. The disadvantages of the approach actually include the domain specific representation, because it necessitates representation specific algorithm operators. Furthermore, these new operators may alter the behaviour of the optimisation algorithm itself, because the assumptions for the algorithm may no longer hold.
Feasibility-based approaches	Approaches focused on solution feasibility consider not only the feasibility of a solution, but also the degree of infeasibility. Optimisation problems are generally unchanged with this approach. However, the solution comparison process is designed to favour feasible solutions first, followed by infeasible solutions ordered by the degree of constraint violation [66]. The *feasibility rules* of Deb [66] have also been refined [47, 272].

Table 2.1: (continued)

Classification	Discussion & Critique
	The benefit of feasibility-based approaches are the simplicity of implementation and good performance [47, 66]. It has been shown that the feasibility rules do suffer from problems with candidate solution diversity [67], necessitating the use of diversity introduction approaches.
Repair-based approaches	Solution repairing approaches ensure that solutions remain feasible by adjusting the solution representations based on predefined, problem specific rules. When it is possible to use repair approaches to correct invalid solutions, an improved optimisation algorithm performance is possible [209, 211]. A concern of repair approaches is that they may introduce bias when repairing solutions. Furthermore, the repair process itself may undesirably affect the optimisation algorithm [261, 302].
Appropriated techniques	Other fields which consider constraint optimisation have developed techniques which may be incorporated into an optimisation algorithm. Within the optimisation algorithm, such techniques are either partially simulated or fully replicated. The considered constraint optimisation techniques from other fields may vary from simplistic mathematical processes [13, 176], fuzzy logic [170, 250, 286, 305] and other population-based optimisation algorithms [16, 36, 59, 72, 84, 108, 109, 239, 242].

Table 2.1: (continued)

Classification	Discussion & Critique
	An advantage provided from techniques originating within other fields is that the techniques have already been shown to be effective and can be incorporated into optimisation algorithms as constraint handling approaches. The main disadvantage, however, is that these techniques are potentially difficult to implement. Furthermore, the operation of these techniques might not be suitable for an iterative optimisation algorithm. The constraint handling techniques from other fields may also interfere with the optimisation algorithm, thereby hindering the ability of the optimisation algorithm to find solutions. More importantly, the additional computational cost associated with such approaches may be too expensive to be considered as a viable technique to incorporate within an optimisation algorithm.

Research [47, 58, 103, 124, 133, 147, 251] proposes different strategies to aid the optimisation algorithm in finding feasible solutions. These strategies are collectively known as constraint handling approaches and are focused on SOSC optimisation problems. Although a broad categorisation of constraint handling is given in table 2.1, it is imporatant to note that the listed approaches are not mutually exclusive. As an example, consider the case where a constraint handling approach restructures the optimisation process into a co-evolutionary one. The co-evolutionary process simultaneously evolves not only better solutions to the optimisation problem, but also the parameters for a penalty function. In this situation both the co-evolutionary approach and the problem reformulation (via the adapting penalty function) are the employed constraint handling approaches.

The following constraint handling approaches are relevant for this thesis:

- **α-constraint**: The constraint handling approach transforms a constrained optimisation problem into a boundary-constrained optimisation problem through two separate steps [271, 273]. Firstly, the *satisfaction level* of optimisation problem constraints, $\mu(\mathbf{x})$, is calculated for a given solution. The satisfaction level is calculated as:

$$\mu(\mathbf{x}) = \min_{i,j}(\mu_{g_i}(\mathbf{x}), \mu_{h_j}(\mathbf{x})) \tag{2.11}$$
$$\forall i = 1, 2, \ldots, n_g$$
$$\forall j = 1, 2, \ldots, n_h$$
$$\mu_{h_i}(\mathbf{x}) = \begin{cases} 1 - \frac{|h_i(\mathbf{x})|}{b_i} & \text{if } |h_i(\mathbf{x})| \leq b_i \\ 0 & \text{otherwise} \end{cases} \tag{2.12}$$
$$\mu_{g_j}(\mathbf{x}) = \begin{cases} 1 & \text{if } g_j(\mathbf{x}) \leq 0 \\ 1 - \frac{g_j(\mathbf{x})}{b_j} & \text{if } 0 \leq g_j(\mathbf{x}) \leq b_j \\ 0 & \text{otherwise} \end{cases} \tag{2.13}$$

  where b_i and b_j are proper positive fixed numbers.

  After determining the satisfaction level for each candidate solution, all the candidate solutions are compared using:

$$(f_1, \mu_1) <_\alpha (f_2, \mu_2) \iff \begin{cases} f_1 < f_2 & \text{if } \mu_1, \mu_2 \geq \alpha \\ f_1 < f_2 & \text{if } \mu_1 = \mu_2 \\ \mu_1 > \mu_2 & \text{otherwise} \end{cases} \tag{2.14}$$
$$(f_1, \mu_1) \leq_\alpha (f_2, \mu_2) \iff \begin{cases} f_1 \leq f_2 & \text{if } \mu_1, \mu_2 \geq \alpha \\ f_1 \leq f_2 & \text{if } \mu_1 = \mu_2 \\ \mu_1 > \mu_2 & \text{otherwise} \end{cases} \tag{2.15}$$

  where the α comparison is defined as an order relation on the set of $(f(\mathbf{x}), \mu(\mathbf{x}))$. The comparison considers a lexicographic order in which

$\mu(\mathbf{x})$ precedes $f(\mathbf{x})$. It should be noted that ensuring the candidate solution $\mathbf{x}$ is a feasible solution is more important than optimising the objective function value, $f(\mathbf{x})$.

The α-constraint may also be used as a penalty method with the following formulation to determine the satisfaction level:

$$\varphi(\mathbf{x}) = \sum_i \| \max\{0, g_i(\mathbf{x})\} \|^p + \sum_j \|h_j(\mathbf{x})\|^p \tag{2.16}$$

$$\mu(\mathbf{x}) = \max\left\{0, 1 - \frac{\varphi(\mathbf{x})}{B}\right\} \tag{2.17}$$

where p and B are positive fixed numbers and $\mu(\mathbf{x})$ is the penalty term that is added to the objective function.

A similar approach is known as the ϵ-constraint [272] which introduces a relaxation to the feasibility rule criterion outlined by Deb [66]. Selecting the best candidate solution based on the set of objective function values and constraint violations is then performed similarly to the α-constraint, where ϵ replaces α within equations (2.14) and (2.15). When $\epsilon = 0$, the ϵ-constraint approach reduces to the original set of feasibility rules defined by Deb [66]. The ϵ-constraint is not considered due to the additional parameter ϵ which requires a user defined value. The value of ϵ is problem dependent, which would require tuning in order to define an optimised value. As a result, the additional ϵ parameter increases the overall complexity of the approach used to solve an optimisation problem.

- **Lagrangian reformulation**: The Lagrangian formulation, similar to the α-constraint, translates a COP into a boundary-constrained optimisation problem. Application of the Lagrangian [256, 270] reformulation produces a *dual problem* which is associated with the *primal problem*, as given in definition 2.2. The dual problem is defined as:

Definition 2.5. *Lagrangian transformation (dual problem)*

$$\begin{aligned} \underset{\boldsymbol{\lambda}, \boldsymbol{\sigma}}{\text{maximize}} \quad & L(\mathbf{x}, \boldsymbol{\lambda}, \boldsymbol{\sigma}) = f(\mathbf{x}) + \boldsymbol{\sigma}^T g(\mathbf{x}) + \boldsymbol{\lambda}^T h(\mathbf{x}) \\ \text{subject to} \quad & \sigma_m \geq 0, \quad m = 1, \ldots, n_g, \\ & g_m(\mathbf{x}) \leq 0, \quad m = 1, 2, \ldots, n_g, \\ & h_m(\mathbf{x}) = 0, \quad m = n_g + 1, n_g + 2, \ldots, n_g + n_h \end{aligned} \tag{2.18}$$

where $\boldsymbol{\sigma}^T$ and $\boldsymbol{\lambda}^T$ are transposed multiplier vectors for the inequality and equality constraints.

The aim of the Lagrangian transformation is to find the saddle-point $\{\mathbf{x}^*, \boldsymbol{\sigma}^*, \boldsymbol{\lambda}^*\}$, such that

$$L(\mathbf{x}^*, \boldsymbol{\sigma}, \boldsymbol{\lambda}) \leq L(\mathbf{x}^*, \boldsymbol{\sigma}^*, \boldsymbol{\lambda}^*) \leq L(\mathbf{x}, \boldsymbol{\sigma}^*, \boldsymbol{\lambda}^*) \tag{2.19}$$

Solving the min-max problem

$$\min_{\mathbf{x}} \max_{\sigma,\lambda} L(\mathbf{x}, \boldsymbol{\sigma}, \boldsymbol{\lambda}) \tag{2.20}$$

provides the minimiser for $\mathbf{x}^*$ as well as the multipliers $\boldsymbol{\sigma}^*$ and $\boldsymbol{\lambda}^*$.

For non-convex optimisation problems, the solution of the dual problem does not match the solution of the primal problem. In such scenarios, a penalty function is added to the Lagrangian problem reformulation:

$$P(\mathbf{x}) = \frac{P}{2}\left(\sum_{i=1}^{n_g}(g_i^{+2}(\mathbf{x}) + \sum_{j=n_g+1}^{n_g+n_h} h_j^2(\mathbf{x}))\right) \tag{2.21}$$

where $P > 0$ is a positive penalty parameter, and

$$g_i^+(\mathbf{x}) = \max(0, g_i(\mathbf{x})), \quad i = 1, \ldots, n_g \tag{2.22}$$

The addition of the penalty term into the formulation of the Lagrangian, produces the *augmented Lagrangian* [98, 270] which is usually written as

$$L_a(\mathbf{x}, \boldsymbol{\sigma}, \boldsymbol{\lambda}, \boldsymbol{\rho}) = f(\mathbf{x}) + \sum_{i=1}^{n_g} p_i(\mathbf{x}, \boldsymbol{\sigma}, \boldsymbol{\rho}) + \boldsymbol{\lambda}^T h(\mathbf{x}) + \boldsymbol{\rho} \sum_{i=n_g+1}^{n_g+n_h} h_i^2(\mathbf{x}) \tag{2.23}$$

where

$$p_i(\mathbf{x}, \sigma_i, \rho) = \begin{cases} \sigma g_i(\mathbf{x}) + \rho g_i^2(\mathbf{x}) & \text{if } g_i(\mathbf{x}) \geq \frac{-\sigma_i}{2\rho} \\ -\frac{\sigma_i^2}{4\rho} & \text{if } g_i(\mathbf{x}) < \frac{-\sigma_i}{2\rho} \end{cases} \tag{2.24}$$

It can be demonstrated that the saddle-point solution, $\{\mathbf{x}^*, \boldsymbol{\sigma}^*, \boldsymbol{\lambda}^*\}$, of the augmented Lagrangian is identical [10] to the solution of the primal problem (in definition 2.2). The method of Lagrange multipliers, from which the Lagrangian approach to constraint handling developed, originates within the field of mathematical optimisation [10].

Based on the work by Barbosa [9], Tahk and Sun [270] applied the augmented Lagrangian approach to solve SOSC problems. The min-max problems were solved using a co-evolutionary evolutionary strategy (ES) approach, which also included an annealing scheme. PSO was later used as the population-based optimisation algorithm within the co-evolutionary formulation by Shi and Krohling [256].

2.3.3 Costs of Constraint Handling

Constraint handling approaches increase the computational cost of the optimisation algorithm. The additional cost should be considered when defining the computational budget. In certain cases, the additional computational expense for a constraint handling approach may be too much for the current budget. This consideration is especially relevant when the underlying optimisation problem changes over time. When including problem level constraints that also change over time, the efficiency of the constraint handling approach may actually hinder the optimisation problem. The degree to which the constraint handling approach possibly interferes with the dynamic optimisation algorithm is not known. However, it is a certainty that the overall complexity of the optimisation process does increase.

For the SOSC scenario, the additional complexity within the optimisation process is limited to the management of the problem constraints. Similarly, SODC and DOSC problems have increased complexity, whilst accounting for the dynamically changing constraints or problem landscape. The increase in complexity is larger for SODC and DOSC when compared to SOSC problems. Lastly, the complexity of DODC problems is the largest, where changes to the problem landscape and the problem constraints may occur independently.

2.4 Conclusion

This chapter introduced the optimisation process and discussed how the process consists of three separate, yet inter-connected parts. Each of the parts of the optimisation process is required to work together in attempts to locate viable solutions to the optimisation problem.

The complexity of the overall optimisation process increases as the definition of the optimisation problem changes. When considering boundary-constrained static optimisation problems, the complexity of the optimisation problem is determined by the evaluation of the objective function and the chosen optimisation algorithm. The addition of problem constraints limits the problem search space within which the optimisation algorithm can find feasible solutions. The search process is guided to more desirable solutions through the use of constraint handling approaches, which increases problem complexity.

Optimisation problems that change over time provide a larger problem complexity, requiring that the optimisation algorithm adapt to the problem as the changes occur. When constraints are included within DOPs, the problem complexity increases even more. The resulting complexity increase becomes an unmeasured maximum when both the problem search page and constraints change over time. DCOPs are complex optimisation problems which may provide a challenge which is more difficult to solve than the challenge to solve dynamic optimisation problems. Although DOPs are not yet fully understood,

DCOPs provide the next challenge from which more capable optimisation algorithms can be produced.

Chapter 3

Population-based Optimisation Algorithms

We cannot solve our problems with the same thinking we used when we created them.

Albert Einstein

Optimisation algorithms are search processes that aim to locate the best possible solution to a given problem as quickly as possible. The search for the optimum solution is an iterative process whereby a candidate solution is modified with the intention of obtaining an even better solution for the given optimisation problem. Deterministic optimisation algorithms are a class of algorithm which always produce the same result for a given set of inputs, just like a pure mathematical function. In contrast, non-deterministic optimisation algorithms may produce different solutions to the given optimisation problem. The varying solutions from these non-deterministic algorithms are a consequence of random variables within non-deterministic algorithms. Furthermore, random variables prevent the predictability of these non-deterministic optimisation algorithms.

Non-deterministic, or *stochastic*, optimisation algorithms rely on the use of random variables and the feedback obtained from the observed candidate solutions to guide and direct the search process through the optimisation problem. Unlike deterministic methods which are often direct methods resulting in a single solution, stochastic algorithms may use multiple candidate solutions at once in order to locate the best possible solution. The focus of this chapter is on a class of non-deterministic optimisation algorithms that use a collection of candidate solutions, modelled after metaphors in biology. By emulating the biological processes, these algorithms display an emergent behaviour which ultimately aids the optimisation algorithm in obtaining a solution to an optimisation problem.

The sections that follow discuss the GA, DE and PSO static optimisation algorithms in section 3.1. Section 3.2 elaborates on the algorithmic considerations for dynamic environments and how static optimisation algorithms may be adapted to cope with changing problem landscapes. The impact of constraint handling methods on a dynamic optimisation algorithm are highlighted in section 3.4 before providing a summary of the chapter in section 3.5.

3.1 Static Optimisation Algorithms

Static optimisation algorithms are a category of optimisation algorithm which focus on provising solutions to static optimisation problems. This section discusses three static optimisation algorithms, modelled after different biological metaphors. Sections 3.1.1 to 3.1.3 respectively briefly discuss the GA, DE and PSO algorithms. These algorithms form the base algorithms that are built upon later within this thesis and are the only considered population-based evolutionary and swarm intelligence algorithms.

3.1.1 Genetic Algorithm

The GA models the process of *natural selection* [57] with the surviving individuals of a species winning the right to propagate their genetic material to subsequent generations.

The sections that follow discuss each of the modelled natural selection processes in more detail. The canonical generational (or iteration based) GA algorithm is given in section 3.1.1.1, and the different behaviours within the algorithm are highlighted. Section 3.1.1.2 discusses different selection schemes for the GA, with section 3.1.1.3 discussing reproduction operators and section 3.1.1.4 discussing different mutation strategies for offspring candidate solutions.

3.1.1.1 Canonical Algorithm

The collection of genes, or *genotype*, determines the final composition of the candidate solution (or *individual*). Through the combination of genes, an offspring individual may override traits that are present in parent individuals. An example of genes in offspring overriding the traits present in parents is the "brown-eye trait" in humans [268], where offspring may develop an iris pigmentation that differs from the iris pigmentation of parents. Darwin also proposed that offspring may acquire additional minor differences when compared to the parents. These differences are called *mutations* and may or may not be beneficial to the offspring, thereby increasing or decreasing the offspring's chances for survival. The theories of Darwin were later substantiated by Fr. Gregor Mendel (1822–1884), who is regarded as the father of the field of genetics.

Algorithm 3.1 Genetic Algorithm

$n_s \leftarrow$Randomly initialise n_x-dimensional candidate solutions
$n_s \leftarrow$ EVALUATE(n_s)
repeat
  parents $\leftarrow$SELECTION(n_s) ▷ *select parents*
  offspring $\leftarrow$REPRODUCTION(parents) ▷ *create offspring*
  offspring $\leftarrow$MUTATE(*offspring*) ▷ *mutate offspring*
  offspring $\leftarrow$EVALUATE(*offspring*) ▷ *evaluate offspring*
  $n_s \leftarrow$ COMBINE(n_s, *offspring*) ▷ *next population*
until stopping condition(s) satisfied

The GA was popularised through the work of Holland [120], even though the previous work of Bremermann [28] and Fraser and Burnell [95] introduced the initial optimisation algorithm formulation. New candidate solutions are generated from the current set of candidate solutions, referred to as the current *generation*, through:

1. **Selection** determines which candidate solutions in the current collection of candidate solutions are allowed to take part in the reproduction process to produce offspring.
2. **Reproduction** describes the process where the genetic material from a collection of parent candidate solutions is recombined to produce offspring candidate solutions.
3. **Mutation** describes an adaptation procedure where new genetic material is introduced to the offspring candidate solutions to increase the genetic diversity of the collection of candidate solutions.

Algorithm 3.1 presents the pseudo-code for the GA. The pseudo-code defines the structure of the algorithm without detailing the specifics for the operators within the GA.

3.1.1.2 Selection

The selection process determines which of the current generation candidate solutions will participate to produce offspring. Repeated selection of the same parent candidate solutions will, over time, cause subsequent generations to become homogeneous (*i.e.*, candidate solutions become more and more similar). The similarity over time is due to the *selective pressure* [8, 102, 155] within selection operators. Larger selective pressure favours the more desirable, or more fit candidate solutions of the current generation. The large selective pressure results in less of the optimisation problem search space being explored as the fitter candidate solutions bias the focus to a specific area of the optimisation problem search space. The selective pressure present in the optimisation

algorithm should be a consideration during algorithm design. If the problem allows for such exploitation, higher selective pressures may aid in producing better solutions more quickly. Characteristics of the problem search space are not always available and selection operators that have lower selective pressure should be considered instead.

3.1.1.3 Reproduction

New candidate solutions are the result of the reproduction process and occurs after selecting parent candidate solutions. The reproduction process defines the required number of parents in order to execute and requires at least one parent candidate solution [280]. A *crossover* operator recombines the genetic material of the parent candidate solutions to produce zero or more offspring candidate solutions. The probability of crossover, $p_c \in [0, 1]$, determines if the crossover operator should occur for a sample of parent candidate solutions. The crossover probability is also known as the crossover rate. Two main categories of crossover operators exist which categorise the reproduction as either *intermediate recombination* (for continuous representation) or *discrete recombination* [202, 280]. Recombination may be as simplistic as exchanging the genetic material of parent candidate solutions between selected pivot points (as in *k-point* and *uniform* crossovers), to more sophisticated approaches like parent centric crossover (PCX) [64, 65] and fuzzy recombination [292].

3.1.1.4 Mutation

Offspring candidate solutions can include new genetic material though *mutation*. When the new genetic material is inserted into the offspring, the genetic diversity within the current generation increases. Each gene within the offspring is mutated at a given probability, $p_m \in [0, 1]$. The probability threshold, p_m, allows for a non-zero probability to not apply any mutation to the current gene. Gene data representations determine the effect of mutation. Continuous-valued representations are mutated by adding a mutational step size to selected gene values. Mutational step sizes are sampled from a probability distribution to add an amount of noise to the genes. The probability distribution from which the noise value is sampled should have a mean value of zero. If the probability distribution does not have a zero mean, repeated sampling of a mutational step size will introduce *genetic drift*, where the sampled mutational step sizes increase over time.

Application of mutation to an offspring does not prevent the resulting candidate solution from becoming less desirable:

- Larger values of p_m encourage exploration of the optimisation problem search space, because more mutations will occur on offspring solutions.

- Smaller values of p_m restrict the search space exploration to the immediate area around the original offspring candidate solution location within the problem search space.

When the focus of the mutation operator is exploration, larger step sizes will facilitate larger jumps throughout the problem search space. On the other hand, larger mutational step sizes may prevent stepping into areas of the problem search space that contain better solutions, by stepping over these areas. It may be beneficial to start the optimisation algorithm search process with larger initial p_m values to allow for more initial exploration of the problem search space, reducing p_m over time to encourage exploitation and the refinement of solutions.

3.1.2 Differential Evolution

Storn [266] and Storn and Price [267] proposed the DE as a stochastic optimisation algorithm for static optimisation problems. The general formulation of the DE algorithm is similar to that of the GA, reusing the p_m and p_c control parameters. The critical difference between the algorithms is the manner in which the DE determines the mutational step sizes which are applied to offspring candidate solutions.

The sections that follow discuss the DE algorithm by defining the structure of the canonical algorithm in section 3.1.2.1. Trial vector creation and reproduction is discussed in section 3.1.2.2, which replaces the mutation operators within the GA and other similar evolutionary algorithms (EAs). The popular DE naming shorthand is described in section 3.1.2.3. Lastly, an effective self-adaptive DE algorithm is discussed in section 3.1.2.4.

3.1.2.1 Canonical Algorithm

Mutational step sizes within the DE are calculated using multiple stochastic, weighted difference vectors that are calculated from randomly selected parent candidate solutions. The resulting weighted vector is known as a *trial vector*. The trial vector then takes part in the offspring creation process in order to produce a single offspring candidate solution. Another notable difference with the GA is that the mutation operator is executed before the reproduction or recombination operator. Once the offspring candidate solution is produced, the candidate solution with the more desirable fitness value survives to be a member of the next generation. Algorithm 3.2 provides pseudo-code for the general structure of the DE.

3.1.2.2 Trial Vector Creation and Recombination

From the current generation distinct candidate solutions are sampled at random and optionally based on specific criteria, such as being the best candidate

Algorithm 3.2 General Differential Evolution Structure

Randomly initialise a n_x-dimensional set of n_s candidate solutions
Evaluate the fitness for all candidate solutions
repeat
  for each $\mathbf{x}_i(t) \in$ candidate solutions **do**
    $\mathbf{u}_i(t) \leftarrow$ CreateTrialVector( )
    $\mathbf{x}'_i(t) \leftarrow$ Crossover$(\mathbf{u}_i(t), \mathbf{x}_i(t))$
    $\mathbf{x}_i(t+1) \leftarrow$ ReplaceFittest$(\mathbf{x}'_i(t), \mathbf{x}_i(t))$
  end for
until stopping condition(s) satisfied

solution of the current generation. The random selection of parent candidate solutions lessens the influence of selective pressure by considering all unique candidate solutions. Information about the optimisation problem is derived directly from the sampled candidate solutions. From these parent candidate solutions a trial vector, $\mathbf{u}_i(t)$, is produced [58, 237, 266, 267] which represents the mutational step size vector, for each individual. The mutational step size approaches a Gaussian (or Normal) probability distribution, based on the central limit theorem [263] provided that sufficient trial vectors [266] are created.

The offspring candidate solution, $\mathbf{x}'_i(t)$, is produced by exchanging dimensions between the trial vector and target candidate solution. Crossover takes place at pre-selected dimensions for the n_x-dimensional candidate solution [266, 267]. The generated offspring individual is then evaluated using the problem objective function to determine its fitness. If the offspring is more fit than the parent, it replaces the parent in the next generation of the algorithm, otherwise it is discarded.

3.1.2.3 Algorithm Naming Schemes

Numerous different strategies for the DE have been developed with these strategies adapting a specific aspect of the DE algorithm. The manner in which the trial vector is created and the reproduction procedure are the most common strategies to vary, resulting in the definition of a naming shorthand for the DE [266, 267]. It is common to refer to the DE using a naming scheme in the form DE/x/y/z. Within this notation, x refers to the trial vector selection strategy, y defines the number of difference vectors used and z determines the crossover operator. Notation examples include:

- **DE/rand/1/bin**: A DE where the trial vector selection is performed randomly with a single difference vector and offspring are created using binomial crossover.

- **DE/best/2/exp**: Trial vector creation is based on the candidate solution with the best objective function value and requires two difference vectors. The offspring candidate solution is determined through the use of the exponential crossover between the trial vector and the parent candidate solution.

3.1.2.4 Self-Adaptive Differential Evolution

Self-adaptive differential evolution (SaDE) [238] was proposed as an extension to the canonical DE algorithm, whereby the control parameters of the algorithm adjust during algorithm execution. The control parameters include the weights for the difference vectors, together with the parameters defining the rates for mutation and crossover. The algorithm maintains different memories to control the amount of previous experience from which the algorithm can estimate new algorithm control parameters. Counting success and failure rates of solution improvement determines the algorithm control parameter update criteria. The only algorithm control parameter which requires the user is the number of algorithm individuals, n_s. SaDE has shown to successfully solve SOPs prepared for the "CEC2005 Special Session on Real-Parameter Optimisation" optimisation algorithm competition [269].

3.1.3 Particle Swarm Optimisation

Kennedy and Eberhart [140] introduced the PSO in 1995. The PSO is a stochastic, population-based search algorithm founded on the metaphor of flocking birds by simulating complex flight patterns. The PSO maintains a *swarm* of homogeneous candidate solutions which fly through a hyper-dimensional problem search space. Within the swarm, candidate solutions are referred to as *particles*.

The sections that follow provide an overview of the PSO in section 3.1.3.1. A PSO variant which self-adapts algorithm control parameters and has shown good performance within static optimisation problems is discussed in section 3.1.3.2.

3.1.3.1 Canonical Particle Swarm Optimisation

PSO describes the movement of the swarm through the search space, by defining the movement characteristics of the individual particles themselves. Although the behaviour of individual particles is simple, the resulting collective behaviour is complex. Eberhart and Shi [81] extended PSO, introducing an inertia coefficient to either reinforce or diminish the contribution of a particle's current trajectory. PSO with the inertia coefficient is the algorithm variant considered as the basic canonical PSO formulation.

Individual particles maintain three pieces of information:

- The current position within the multi-dimensional problem search space.
- The best, previously observed, position since the start of the algorithm.
- A velocity vector determining the direction and magnitude of movements within the problem search space.

Using this information, each particle adheres to the following set of behaviour rules:

1. Move towards the best, previously observed position.
2. Move towards the closest and best neighbouring particle.
3. Continue on the current movement trajectory (inertia).

PSO uses an iterative process to move particles through the problem search space. Particle movement requires the calculation of a velocity vector, $\mathbf{v}_i(t+1)$, at each iteration and is remembered by the particle. The velocity update consists of three terms of influence or *particle attractors* and is defined as:

$$\begin{aligned}\mathbf{v}_i(t+1) = \omega \cdot \mathbf{v}_i(t) \\ + c_1 \cdot \mathbf{r}_{1i}(t) \cdot [\mathbf{y}_i(t) - \mathbf{x}_i(t)] \\ + c_2 \cdot \mathbf{r}_{2i}(t) \cdot [\hat{\mathbf{y}}_i(t) - \mathbf{x}_i(t)]\end{aligned} \tag{3.1}$$

where each of the terms within equation (3.1) serves to fulfil the behaviour rules of the particle. Each particle attractor describes a specific particle movement characteristic:

- The **inertia component** $(\omega \cdot \mathbf{v}_i(t))$ determines the tendency of the particle to remain on its current movement trajectory. Values of $\omega \in [0, 1)$ reduce the weighting of the previous velocity within equation (3.1), whereas $\omega \geq 1$ reinforces the weight of the previous velocity vector.
- Each particle tries to move towards their best, previously observed position through the use of the **cognitive component** $(c_1 \cdot \mathbf{r}_{1i}(t) \cdot [\mathbf{y}_i(t) - \mathbf{x}_i(t)])$. A constant value, c_1, is a scaling factor which is multiplied with a stochastic vector of values that are sampled from a uniform distribution, $\mathbf{r}_{1i}(t) \sim U_{n_x}(0, 1)$ to perturb the cognitive component.
- The influence of the closest, neighbourhood best position, $\hat{\mathbf{y}}_i(t)$, is given by the **social component** $(c_2 \cdot \mathbf{r}_{2i} \cdot [\hat{\mathbf{y}}_i(t) - \mathbf{x}_i(t)])$. Similar to the cognitive component, c_2 is a constant scaling factor that is multiplied with a distinct stochastic component $\mathbf{r}_{2i}(t) \sim U_{n_x}(0, 1)$ to scale and perturb the social component.

  Social connections between particles are determined by social structures or topologies. The more interconnections between particles, the more

particles are attracted to the current "best" particle within the local neighbourhood. Different topology structures exist [134, 162], with the most frequently used topologies being the *star*, *ring* and *Von Neumann* structures. Alternative names for the star and ring neighbourhood topologies respectively are the *global best* (or *gbest*) and *local best* (or *lbest*) neighbourhood topologies. The Von Neumann topology has been empirically shown to perform well on a large collection of problems [84, 106, 139]. No one topology is definitively "better" than another, with topology choice being problem dependent [162] and an influence on the computational budget [84, 141].

The acceleration coefficients, c_1 and c_2, control the balance of the particle movement. When $c_1 > c_2$, the particle will favour its own personal best position instead of the best position close to the current particle position. Conversely, when $c_1 < c_2$ the social information provided to the particle will dominate the particle velocity calculation. The extreme case is when $c_1 = 0$ or $c_2 = 0$, where the particle will respectively either only consider its own personal experience or the social experience, foregoing the other. Generally, the coefficients should be $c_1 \approx c_2$ so that the social and personal experience provide a decent amount of both exploitation and exploration, controlled respectively by c_1 and c_2. Additionally, larger values for the acceleration coefficients will create larger step sizes within the problem search space. When the step sizes are larger, the chances of "jumping over" an optimum increase, and reduce when the step sizes are lower. The best values of the acceleration coefficients are, unfortunately, problem dependent. To prevent overly large step sizes for particles, the resulting velocity vector may be "clamped". By clamping velocity vector dimensions, a limit is enforced on the resulting step size for each dimension by defining limits. Furthermore, it has been shown that particles have a roaming like behaviour [86], where particles move away before returning to the swarm. As a result, clamping particle velocities may impede the roaming particle behaviour and can be regarded as a repair strategy (see table 2.1).

Given the newly calculated velocity vector, $\mathbf{v}_i(t+1)$ for particle i, the movement of the particle at time-step t to the next position within the problem search space is calculated as:

$$\mathbf{x}_i(t+1) = \mathbf{x}_i(t) + \mathbf{v}_i(t+1) \tag{3.2}$$

As a final step in the update process of a particle, the memory of the particle is updated to record the best observed candidate solution thus far. If the new candidate solution is more desirable than the currently remembered, best candidate solution, the memory of the best candidate solution is updated as follows (assuming minimisation):

$$\mathbf{y}_i(t+1) = \begin{cases} \mathbf{x}_i(t+1) & \text{if } f(\mathbf{x}_i(t+1)) < f(\mathbf{y}_i(t)) \\ \mathbf{y}_i(t) & \text{otherwise} \end{cases} \tag{3.3}$$

Algorithm 3.3 Basic Synchronous PSO

Create and initialise an x_n-dimensional swarm
repeat
  for each particle $i = 1, 2, \ldots, n_s$ **do**
    if $f(\mathbf{x}_i)$ is better than $f(\mathbf{y}_i)$ **then**
      Assign $\mathbf{y}_i = \mathbf{x}_i$
    end if
    if $f(\mathbf{y}_i)$ is better than $f(\hat{\mathbf{y}})$ **then**
      Assign $\hat{\mathbf{y}} = \mathbf{y}_i$
    end if
  end for
  for each particle $i = 1, 2, \ldots, n_s$ **do**
    update velocity using equation (3.1)
  end for
  for each particle $i = 1, 2, \ldots, n_s$ **do**
    update position using equation (3.2)
  end for
until *stopping condition is true*

The PSO pseudo-code is provided in algorithm 3.3.

3.1.3.2 Gaussian-Valued Particle Swarm Optimisation

Harrison [115] proposed an alternative PSO algorithm based on an investigation of PSO control parameter stability. Within the study, multiple PSOs were compared and considered, based on the manner in which the algorithms managed, controlled and adapted PSO control parameters. The investigation considered the bare-bones particle swarm optimisation (BBPSO) [138] algorithm as an almost parameter-less PSO. BBPSO moves particles through the problem search space by assigning new positions based on a bell-curve which is centred around the midpoint between the global and particle best positions. The result is that equations (3.1) and (3.2) are not used within the BBPSO definition, rendering the algorithm as "something else" when compared to PSO variants. From this conclusion, BBPSO was not considered within the PSO parameter study of [115].

The Gaussian-valued particle swarm optimisation (GVPSO) [115] was proposed as new PSO variant that maintained the favourable parameter-less nature of the BBPSO without losing the PSO identity. GVPSO updates the position of a particle by sampling a Gaussian distribution in order to control the step size of the particle as it moves through the problem search space. The distances between the current particle position and the personal best and neighbourhood best positions are used within the calculation of the step size vector, referred to as the ancillary position, $\Delta_{ij}(t)$. The new position of the particle is then

determined using a Gaussian distribution that is centred between the current and ancillary particle positions. The standard deviation of the distribution is based on the magnitude of the distance between the current position and $\Delta_{ij}(t)$ according to:

$$x_{ij}(t+1) = \begin{cases} y_{ij}(t) & \text{if } U(0,1) < e \\ \mathcal{N}\left(\frac{x_{ij}(t)+\Delta_{ij}(t)}{2}, |\Delta_{ij}(t) - x_{ij}(t)|\right) & \text{otherwise} \end{cases} \tag{3.4}$$

such that

$$\Delta_{ij}(t) = x_{ij}(t) + r_{1ij}(t)(y_{ij}(t) - x_{ij}(t)) + r_{2ij}(\hat{y}_{ij}(t) - x_{ij}(t)) \tag{3.5}$$

where e is a user-defined parameter controlling the degree to which the personal best position is exploited. It was shown that the GVPSO produced favourable results within SOPs and outperformed a variety of other PSO algorithm variants, including self-adaptive PSO variants.

3.2 Dynamic Optimisation Algorithms

When a static optimisation algorithm is applied to a dynamically changing optimisation problem, the inefficiencies of the optimisation algorithm become evident. The inefficiencies include the loss of candidate solution diversity, outdated memory values due to the changing optimisation problem and the algorithm's inability to determine that the optimisation problem search space has changed [265]. As a result, a different class of optimisation algorithm developed in order to cope with the changing problem landscape.

Section 3.2.1 introduces different approaches that optimisation algorithms have used to cope with the changing problem landscape. The remainder of this section discusses algorithms that have been effective within dynamic environments. These algorithms include modifications to the GA, DE and PSO.

3.2.1 Dynamic Optimisation Approaches

Dynamic optimisation algorithms aim to not only obtain the best possible solution, but to also track and to maintain solutions present within the dynamically changing problem search space. This class of optimisation algorithm also needs to learn as much as possible from previous search experiences to hopefully aid in the search process.

The sub-sections that follow discuss the broad classification of approaches used by optimisation algorithms within dynamically changing search spaces. The classifications are not mutually exclusive and a dynamic optimisation approach may exist within multiple categories. Section 3.2.1.1 discusses how

optimisation algorithms may determine that a search space change has occurred, whereas sections 3.2.1.2 and 3.2.1.3 discuss diversity management within the dynamic optimisation algorithm. Memory, prediction and adaption approaches are discussed in sections 3.2.1.4 to 3.2.1.6, with the use of multiple candidate solution populations discussed in section 3.2.1.7.

3.2.1.1 Change Detection

Optimisation algorithms are usually not aware of changes to the problem landscape. Algorithms should either detect the presence of a change themselves during execution or require that external information be given stating that a change has occurred. Change detection occurs in one of two ways:

1. Assign specific candidate solutions to be *change detectors*. An optimisation algorithm may then use the change detectors to determine if the underlying optimisation problem search space has changed. The change detectors are either specific candidate solutions, predefined fixed-points in the problem search space, or possibly a set of random candidate solutions [32, 244].

   Importantly, the change detectors (also known as *sentries*) are usually not part of the candidate solution set, requiring that the algorithm maintain the detectors separately. This does not, however, preclude the candidate solutions from participating in the optimisation process. Any difference in objective function value in subsequent optimisation algorithm iterations is indicative of a change in the problem landscape. Based on the detected change, an optimisation algorithm may respond with an appropriate mechanism. A drawback to using change detectors is the additional objective function evaluations incurred by the algorithm to determine if an optimisation problem search space did change. Conversely, the use of sentries creates a robust, dynamic algorithmic behaviour which can react to optimisation problem search space changes.

2. Algorithms may also examine the candidate solutions to determine if the problem search space has changed. If the average objective function value for a number of candidate solutions drops over a period of time (usually algorithm iterations), a change in the problem landscape might have occurred. Interrogating the candidate solutions in every iteration does not necessarily indicate that an optimisation problem search space change has occurred, nor that the interrogation process is itself foolproof. Additionally, the response of the optimisation algorithm may be delayed. An optimisation algorithm needs to observe enough evidence of a change before reacting. For example, a statistical hypothesis test may be used as the measure to determine if a reaction should execute [244]. Furthermore, the reaction of an optimisation algorithm to a change may

be unnecessary [208, 244]; the landscape could change once more whilst the optimisation algorithm is responding to the previous problem search space change.

3.2.1.2 Diversity Introduction

Over multiple iterations of an optimisation algorithm, candidate solutions become more similar as they convergence to a point in the problem search space. Increased similarity of solutions reduces the dispersion of solutions across the problem search space. With lower dispersion, the exploration of other regions within the optimisation problem search space is not possible. Therefore, a reduced exploration lowers the likelihood of detecting new optima in those areas after a problem landscape change.

One approach to increase (or introduce) diversity is to modify candidate solutions upon detecting a change in the problem search space. Within optimisation algorithms that use a mutation operator, the mutation probability can be increased temporarily to introduce more diversity. Cobb [43] suggested that the rate of mutation within the evolutionary algorithm be increased by a predefined "hyper-mutation" factor. The state of hyper-mutation would continue until a threshold is reached. The simplest strategy would be to allow the hyper-mutation to continue for the following n iterations of the algorithm before reducing the mutation rate to the original level. More complex strategies are also available, such as defining sentry positions within the problem search space that are near feasible region boundaries [206]. Upon a landscape change, if the sentry had transitioned from feasible to infeasible space, the hyper-mutation would start and continue until successfully locating a feasible region. After locating a feasible region of the problem search space once again, the mutation rate would return to the "normal" value.

Riekert *et al.* [246] re-introduced diversity in genetic programming (GP) by not only increasing the mutation rate within the algorithm, but by also reducing the percentage of elite individuals that would survive into the next generation. This continued until a satisfactory amount of diversity was once again present between all candidate solutions.

In [189], diversity is introduced by assigning different strategies to candidate solution sub-populations. The strategies include the re-initialisation of an entire sub-population, relocating a candidate solution around a best candidate solution through a quantum process [17] by using Brownian initialisation or through the addition of noise sampled from a Gaussian distribution.

In the context of PSO, Eberhart and Shi [81] suggested that the entire swarm should be re-initialised, or at least a percentage of the swarm should re-initialise. Ignoring immediate re-initialisation and allowing the search process to continue is also valid, provided that the search space changes are small and that enough diversity is still present within the swarm. The best candidate solution should remain unchanged, allowing the continued tracking of the best

solution found so far. Janson and Middendorf [127] subdivided the particle swarm into multiple sub-swarms which reduced the rate at which convergence to old optimum values would occur after a problem landscape change. The sub-swarms would then once again merge into a single swarm after updating older particle positions to recent values within the changed problem search space.

Even though such strategies allow for the reintroduction of diversity into the candidate solutions, the trigger to perform the diversity introduction is not obvious. The optimisation algorithm still requires the means to observe a search space change and to then start the diversity introduction. Furthermore, when additional control parameters are introduced into the algorithm (such as the hyper-mutation factor) it is unclear as to what value such control parameters should take (problem dependence). Lastly, even though the diversity introduction does increase the diversity of the candidate solutions, the process does not necessarily maintain any previous experience into subsequent algorithm iterations.

3.2.1.3 Diversity Maintenance

An alternative strategy to introducing more diversity into candidate solutions is to rather prevent the loss of diversity. As a result, the diversity of the candidate solutions within an algorithm should be maintained over iterations instead of reducing. Furthermore, by constantly ensuring that the diversity level of the candidate solutions does not reduce, the algorithm need not attempt to determine when a change in the search space has occurred. Due to the maintained diversity the algorithm will naturally adapt to changing problem landscapes. Typically, methods such as random immigrants [104], population-based incremental learning [310], variations on PSO and multi-objective optimisation for dynamic problem landscapes have been used to maintain diversity.

The random immigrants approach employs a simple strategy to maintain diversity. At the end of the algorithm iteration, a number of randomly generated candidate solutions are included into to the current generation. The immigrant solutions are evaluated by the objective function, with the more desirable candidate solutions surviving into the next algorithm iteration.

Candidate solutions may also be rewarded for being genetically different from parent candidate solutions [296]. Within this approach three populations of candidate solutions are maintained. Within the one population, parents are selected based on a distance metric which determines the genetic difference between candidate solutions. Maximising the distance from parent candidate solutions promotes diversity, where the distance metric may be as simple as the Hamming distance between candidate solutions. Within the second population, candidate solutions compete based on the improvement of the objective function value, promoting exploitation of the problem search space. The final population, which is the "normal" population within a GA, represents

the actual solutions to the problem search space. All three populations are considered by the algorithm with the sizes of the additional populations being adjusted in order to react to changes in the underlying problem search space. A disadvantage of this approach is the additional memory required to represent three different candidate solution populations. Moreover, there is an additional computational cost to evaluate the objective function and distance metric.

Blackwell and Branke [17, 18] and Blackwell and Bentley [19] suggested a variation of PSO where particles maintain a *repulsive* force within the charged particle swarm optimisation (CPSO) algorithm. The idea for the strategy modelled the repulsion forces that are present within the atom which prevents particles from being too close to each other, maintaining diversity within the swarm of particles.

Diversity may also be maintained through the use of multiple objectives. For example, with the algorithm formulation of Bui *et al.* [29], two objectives are used to represent the problem objective function and the candidate solution diversity. The algorithm may also be stated as a multi-objective algorithm of more than two objectives [277], where the additional objectives aim to address other requirements of the optimisation problem.

By ensuring that the diversity between candidate solutions remains high, the optimisation process is slowed down [130], requiring more iterations to settle onto a solution. It has also been noted that even with a greater diversity throughout the optimisation process, optimisation algorithms may struggle to maintain solutions when there are small landscape changes. The stochastic elements that maintain the diversity of candidate solutions tend to shadow the effects [44] of the small landscape changes and require larger changes in the problem landscape to observe the change.

3.2.1.4 Memory

Regardless of optimisation algorithm, each candidate solution maintains at least one piece of memory. At minimum, the current search space location for the candidate solution and the associated objective function value are remembered. Particles within the PSO maintain additional memory items which include the previous best search space location and a velocity vector to determine the movement trajectory. When the problem search space changes, the memory information of a candidate solution no longer necessarily reflects the changed problem search space:

1. A previous problem search space determined the objective function value for a candidate solution.

2. Any additional memory items, including information derived based on memory elements, also reflect the previous problem search space.

Due to the possibility of additional memory items within a candidate solution (as required by the optimisation algorithm), there is no single memory update strategy for all algorithm configurations. Each algorithm requires a customised procedure to correctly manage the memory update process for the candidate solutions after the problem search space changes. For example, particles within the PSO reference the best previously observed search space position. Although the position is still within the problem search space, the objective function value for the position may have changed with the search space. Furthermore, the current trajectory of the particle derives from the particle's previous experience within the previous problem landscape. A combination of these memory values should be adjusted when the search space changes, but not all adjustments may be necessary for the current optimisation problem. The most severe memory update would be a complete re-initialisation of all candidate solutions, effectively restarting the optimisation process from zero knowledge.

The memory management of the optimisation algorithm is either implicit or explicit:

- **Implicit memory** within an algorithm is encoded directly into the candidate solution. When a candidate solution uses this encoding scheme, the redundant memory values seem to blend together with the search space position from the perspective of the algorithm. More than a single copy of redundant memory may be maintained by the candidate solution. Usually, a dominance relation [158] is enforced within the individual dimensions of the candidate solution. The dominance relation selects the most appropriate memory value for a dimension and is then presented to the algorithm, based on the underlying problem search space and landscape changes. Lewis *et al.* [158] suggested that multiple copies of redundant memory provide better algorithm performance within dynamic environments.

- **Explicit memory** is managed by the algorithm as a separate operation. The algorithm should define the management and expectation of memory based on the following three conditions:

    1. Content of the memory: The memory of the algorithm may either be treated as a direct memory or as an associative memory store. Direct memory maintains a list or archive of previous desirable candidate solutions [24, 56, 310]. On the other hand, associative memory is used to associate specifics with the algorithm based on a common association reference [82]. There is no restriction on the type of information that may be remembered with associative memory. Within dynamic environments, the current time interval may be used to index the associative memory and may include specific characteristics of the problem search space. Examples of

search space details could include different environmental states, observed patterns and possible state transitions for the problem landscape. Within the PSO, the previous best and velocity vectors for each particle are associated memory values.

2. How memory updates: The decision to replace memory is dependent on the optimisation algorithm. The decision of how the memory updates depends on the type of the memory used within the optimisation algorithm. The update may be based on the age of the memory where older memory values are replaced after a certain time period in order to remain relevant [82, 279]. Alternative options include replacing memory in order to maintain the required amount of diversity or when the objective function value is no longer desirable [24, 82].
3. When memory updates: The updates should occur either when the search space changes, or at a regular interval. When it is not possible to know with certainty that the problem landscape has changed, the memory may update after a certain number of iterations.

The primary purpose of memory within a dynamic optimisation algorithm is to bias the search process. Memory will ideally take advantage of certain characteristics within the optimisation problem to improve the search for optima. Branke [23] highlighted that the use of memories provides little value to the dynamic optimisation algorithm when the optimisation problem presents no recurrence characteristics.

3.2.1.5 Prediction

Prediction approaches aim to use problem characteristics in order to build a forecasting model. The forecasting model can determine when certain patterns will be present within the search space. Although memory based approaches can achieve a similar result, prediction based approaches attempt to cater for more repetition types than what a memory based approach generally could. The major disadvantage of the prediction approach is that the approach is only viable when it is known that the problem landscape will present patterns. In the event that the search space does not have this kind of behaviour, the approach may become a hindrance to the optimisation process by erroneously suggesting locations where optima will be.

In order to build the prediction model, the correct quantity of problem search space data is needed. If this data is available, different machine learning models can process the data and then provide information to the optimisation algorithm, supporting the search process. Regression [259] models, Kalman filters [248] and Markov chains [258] are some of the possible models that may inform the optimisation algorithm. This further highlights the difficulty of prediction based approaches where the models may be inferior when the

available data is sparse, does not match the problem, or when the problem is too stochastic.

3.2.1.6 Self-Adaption

As the optimisation problem changes the algorithm may adapt some of its control parameters (and therefore algorithmic behaviour) to cope with the changing problem search space. Ursem [283] extended the candidate solution to include the algorithm control parameters. Although this technique allows for the control parameters to evolve along with the candidate solution, the results are not always better. For example, Cobb [43] reports that such strategies are still no better than using hyper-mutation, which triggers after the problem landscape changes. The poorer algorithm performance may be attributed to the "curse of dimensionality" [144] where the additional control parameter dimensions become less important, or even "lost", as the dimensionality of the problem increases.

Evolutionary programming (EP) and ES have been of interest to researchers due to the algorithms being able to adapt their own mutational step sizes [6]. Angeline [3] examined self-adaptive EP and identified that the algorithm is not effective for the tested benchmark problems. Weicker [298] questioned the Gaussian mutation in the standard ES self-adaption process, stating that it may not be appropriate for dynamic optimisation.

Self-adaption of step sizes has also been demonstrated for cultural evolutions (CEs) in the work of Saleem and Reynolds [252]. As the belief space of the CE changes over time, the problem search space changes are also recorded within the belief space (amongst other beneficial information). Using the belief space knowledge, the step sizes for CE can adjust dynamically throughout the entire algorithm execution.

Mendes and Mohais [189] developed the dynamic differential evolution (DynDE) algorithm, allowing the DE to operate within dynamic environments. Within the DynDE the control parameters of DE are adjusted during algorithm execution and do not require an optimal initial value.

With the number of possible changes within a dynamic optimisation problem, self-adaption of an optimisation algorithm is the only feasible strategy for algorithm control parameter selection [6, 22, 114, 136, 212]. Obtaining the best performing parameters for a dynamic optimisation algorithm is therefore not beneficial to the optimisation process beyond the initial problem landscape [112, 156].

3.2.1.7 Multiple Populations

An optimisation algorithm may subdivide the current population of candidate solutions into smaller populations. The optimisation algorithm can then focus on different aspects of the optimisation problem within each sub-population.

As a consequence of multiple populations, the total diversity of the candidate solutions can generally be maintained or increased by limiting the exposure to which candidate solutions are within the sub-populations.

Island GAs [105, 154, 303] consist of multiple populations which are simultaneously evolved. Each of the populations are considered to reside on a separate island that is disconnected and independent. Within the islands themselves, the canonical GA processes (*i.e.*, selection, crossover and mutation) continue but only consider candidate solutions of the local island population. Diversity of the islands increases by allowing candidate solutions from other islands to migrate based on a "migration policy" [31, 154]. Migration policies may refuse migrations, allow all migrations or permit some other suitable policy combination. Within dynamic environments the selection of candidate solutions to migrate and where the solutions should migrate to are determined in a probabilistic manner and constrained by the migration policy.

The DynDE, as described in [74], uses a number of different sub-populations where each of the sub-populations maintain and track a single optimum. At the end of an algorithm iteration, the sub-populations within DynDE are compared against each other in a pairwise manner. If the best candidate solutions from each sub-population are considered to be "too close", the best candidate solution of the sub-population with the less desirable objective function value is re-initialised. The proximity of candidate solutions is determined using a distance metric, such as the Euclidean distance measure. A drawback of the DynDE is that the number of optima present within the optimisation problem should be known in advance to determine the number of sub-populations. The number of optima is not always known, especially when problem landscape changes could change the number of search space optima.

Multi-swarm PSOs [17] have also been proposed which maintain and track the optima of an optimisation problem within each sub-swarm. When using the CPSO within the multi-swarm approach the repulsion of particles occurs in a manner such that particles will repel each other, even when in different sub-swarms. Similarly, the quantum hybrid model mentioned in [17] for the multi-swarm approach allows each sub-swarm to individually to track optima. The multi-swarm approach has been applied to artificial bee colony (ABC) in [128].

Niching algorithms [164] sub-divide the population of candidate solutions into multiple sub-populations. Each sub-population maintains a singular problem optima within a multi-modal optimisation problem. The process of creating and maintaining niches allows for candidate solution populations to split into smaller sub-populations when locating new optima that satisfy the algorithm split criteria. Overlapping sub-populations may also merge based on the distance between the sub-populations. Merging sub-populations maintains a single niche optimum more effectively.

Co-evolutionary algorithms (CoEAs) use multiple populations which compete against or cooperate with each other in order to "improve". Competitive

CoEAs [194, 207] focus on evolving populations that challenge one another in order to produce the best possible solution. The competition is usually initiated without any previous knowledge about the problem or any information about the problem. When cooperation is used for the co-evolutionary process, the sub-populations collectively attempt to solve the problem, with the result of cooperation being beneficial, harmful or benign to the sub-populations. In [163, 235, 284], the optimisation problem is divided into smaller parts which are then later combined in order to cooperatively produce a final candidate solution. Hybridised models may also be considered where both cooperation and competition are used within the resulting algorithm [101].

The use of multiple populations is an attractive option. By using multiple populations, more than a single optimum within the problem search space can be located and tracked separately. This effectively creates a parallel search of the problem search space by multiple DE algorithms. An important consideration with such an approach is the effect on the available computational budget. If each sub-population has only a handful of candidate solutions, the search may not be effective enough, diluting the search capability of the algorithm. On the other hand, with an increased number of candidate solutions in each population the computation budget may quickly become exhausted.

3.2.2 Hyper-mutation Genetic Algorithm

Cobb [43] proposed hyper-mutation to allow the GA to introduce diversity within a dynamically changing problem landscape. The canonical GA is known to suffer diversity loss due to the selective pressure present within the algorithm operators. Hyper-mutation is a strategy that allows the GA to switch from the standard mutation rate, p_m to a hyper mutation rate, p_{hyper}, which is a much higher rate value.

The hyper-mutation genetic algorithm (HyperM) uses a simple change detection strategy whereby any degradation in the objective function value enables hyper-mutation. The change detection within HyperM is focused on tracking the objective value of the last known optimum. The hyper-mutation continues for a defined period, which is usually a number of algorithm iterations, before switching back to the standard mutation rate.

Although the HyperM uses a simple method to introduce diversity into the population, it has been shown to be effective. The pseudo-code for the HyperM algorithm is given in algorithm 3.4.

3.2.3 Random Immigrants Genetic Algorithm

Instead of introducing diversity into the current candidate solution population, Grefenstette [104] proposed a method to ensure that the diversity between candidate solutions is always maintained. The random immigrants genetic algorithm (RIGA) is merely an extension of the canonical GA which includes

Algorithm 3.4 Hyper-mutation Genetic Algorithm

Pre: $p_{hyper} \in (p_m, 1.0)$
  $n_s \leftarrow$Randomly initialise n_x-dimensional candidate solutions
  $M_{norm} \leftarrow$ CreateMutateFunction(p_m)
  $M_{hyper} \leftarrow$ CreateMutateFunction(p_{hyper})
  $M_{current} \leftarrow M_{norm}$
  $hyper_{count} \leftarrow 0$
  $hyper_{total} \leftarrow 5$
  $f_{best} = undefined$
  repeat
    $n_s \leftarrow$ Evaluate(n_s)
    $f_{test} \leftarrow$ MostFit(n_s) ▷ *most fit candidate solution*
    if $f_{best} = undefined$ **then**
      $f_{best} = f_{test}$
    end if
    if f_{test} is less fit than f_{best} **then** ▷ *environment changed*
      $M_{current} \leftarrow M_{hyper}$
    end if
    if $hyper_{count} > hyper_{total}$ **then** ▷ *stop hyper-mutation*
      $M_{current} \leftarrow M_{norm}$
      $hyper_{count} \leftarrow 0$
    end if
    $offspring \leftarrow$Crossover(n_s) ▷ *create offspring*
    $offspring \leftarrow M_{current}(offspring)$ ▷ *mutate offspring*
    $offspring \leftarrow$Evaluate($offspring$) ▷ *evaluate offspring*
    $n_s \leftarrow$ Combine(n_s, $offspring$) ▷ *next population*
    $f_{best} \leftarrow$ MostFit(n_s) ▷ *most fit candidate solution*
    if $hyper_{count} < hyper_{total}$ & $M_{current} = M_{hyper}$ **then**
      $hyper_{count} \leftarrow hyper_{count} + 1$
    end if
  until stopping condition(s) satisfied

Algorithm 3.5 Random Immigrants Genetic Algorithm

$n_s \leftarrow$Randomly initialise n_x-dimensional candidate solutions
repeat
  $n_s \leftarrow$ EVALUATE(n_s)
  offspring $\leftarrow$CROSSOVER(n_s) ▷ *create offspring*
  offspring $\leftarrow$MUTATE(*offspring*) ▷ *mutate offspring*
  offspring $\leftarrow$EVALUATE(*offspring*) ▷ *evaluate offspring*
  $n_s \leftarrow$ COMBINE(n_s, *offspring*) ▷ *next population*
  immigrants $\leftarrow$ GENERATE($p_{im} * |n_s|$) ▷ *generate immigrants*
  immigrants $\leftarrow$EVALUATE(*immigrants*) ▷ *evaluate immigrants*
  $n_s \leftarrow$ COMBINE(n_s, *immigrants*) ▷ *diversify next population*
until stopping condition(s) satisfied

an additional algorithm operator as the final step of the algorithm before determining the next population. The final operation introduces immigrants which represent a portion of the current population. These immigrants are randomly generated solutions that add to the current generation. Through the continual introduction of immigrant solutions, the algorithm is able to maintain diversity. As a consequence of the diversity within the population, the algorithm is able to cope with the changing problem landscape. The replacement rate, $p_{im} \in [0, 1)$, defines the proportion of the population for which immigrants should be generated. The pseudo-code describing the behaviour of the RIGA algorithm is provided in algorithm 3.5.

3.2.4 Dynamic Differential Evolution with Combined Variants

Ameca-Alducin *et al.* [2] proposed a variant of the DE for DOPs called dynamic differential evolution with combined variants (DDECV). The algorithm can alternate between two different execution strategies based on changes to the problem search space. The first strategy is designed to ensure that the algorithm performs exploration within the problem search space, whilst the other attempts to focus the algorithm on exploiting and refining current solutions. The exploration strategy is defined as *DE/best/1/bin*, whilst the exploitation strategy is *DE/rand/1/bin*. In a manner much like that of the HyperM algorithm, as soon as a change with the problem search space has been detected, the algorithm strategy is altered. Exploration is promoted for a predefined number of algorithm iterations before switching the algorithm strategy back to the exploitation strategy.

In order to aid in the maintenance of diversity within the algorithm, a portion of the population is replaced with randomly generated immigrant candidate solutions, at the end of each algorithm iteration. The introduction

of random immigrants also operates at two different levels, determined by the current algorithm strategy. After a problem landscape change, an increased number of random immigrants are added to the population. A *normal* level of immigrant introduction resumes when the exploitation strategy replaces the exploration strategy.

Lastly, a local hill-climber is applied to a randomly selected candidate solution, $\mathbf{x}_{rand}(t)$, within the population. A small amount of noise, $\delta \in U(0,1)$, is both added and subtracted from a randomly selected dimension within $x_{rand,i}(t)$. The addition and subtraction of δ produces two neighbouring candidate solutions. After evaluating the objective function for the two neighbouring candidate solutions, the solution with the best objective function value is remembered. This hill-climbing process repeats for a predefined number of times, where the remembered candidate solution is the most fit candidate solution produced by the hill-climbing process. The final result of the hill-climbing process replaces the least fit candidate solution within the population.

The DDECV is therefore a sequence of different mechanisms to cope with problem landscape changes. The performance of the algorithm is favourable but the authors have not determined if all of the problem search space change mechanics are necessary, nor if these methods negatively interfere with the behaviour of the DE. The pseudo-code for the DDECV is provided in algorithm 3.6.

3.2.5 Quantum Particle Swarm Optimisation

Blackwell and Branke [17, 18] defined a variant of PSO for changing problem landscapes with the QPSO. QPSO is a simpler and computationally less expensive version of CPSO. The algorithm takes inspiration from the quantum model of the atom where a central nucleus is surrounded by orbiting electrons. The quantum atomic model used within the QPSO replaces electrons orbiting a nucleus with a probability distribution centred at the nucleus, referred to as the "quantum cloud".

The quantum model splits the swarm of particles into two distinct subgroups. The first group consists of *neutral* particles, while the second subgroup contains *quantum* particles. Quantum particles do not repel each other and move randomly within the quantum cloud which is situated around the current global best particle. The movement of the quantum particles is determined by sampling the probability distribution which represents the quantum cloud. The quantum cloud is a multi-dimensional hyper-sphere with radius r_{cloud}. QPSO maintains diversity within the swarm through the random movement of the quantum particles. The particle movement for quantum particle i, in dimension j is

$$x_{ij}(t+1) \sim P(\hat{y}_{ij}(t), r_{cloud}) \tag{3.6}$$

where P is any chosen probability distribution with mean of $y_{ij}(t)$ and devi-

Algorithm 3.6 Dynamic Differential Evolution with Combined Variants

Pre: $immigrants_{normal}$: number of immigrants before the search space change
Pre: $immigrants_{explore}$: number of immigrants after the search space change
Pre: ILS: number of iterations for local search
  $n_s \leftarrow$Randomly initialise n_x-dimensional candidate solutions
  $DE_{normal} \leftarrow \text{DERAND1BIN}(p_c = 0.8399, \beta = 0.9644)$
  $DE_{explore} \leftarrow \text{DEBEST1BIN}(p_c = 0.8399, \beta = 1.0820)$
  $DE_{current} \leftarrow DE_{normal}$
  $immigrants_{current} \leftarrow immigrants_{normal}$
  $threashold_{explore} \leftarrow 16$
  $threshold_{count} \leftarrow 0$
  $t \leftarrow 0$
  $n_s \leftarrow \text{EVALUATE}(n_s)$
  repeat
    $f_{best,1} = \text{MOSTFIT}(n_s)$
    $n_s \leftarrow \text{EVALUATE}(n_s)$
    $f_{best,2} = \text{MOSTFIT}(n_s)$
    if $f_{best,2}$ is less fit than $f_{best,1}$ **then** ▷ *search space changed*
      $DE_{current} \leftarrow DE_{explore}$
      $threshold_{count} \leftarrow 0$
      $immigrants_{current} \leftarrow immigrants_{explore}$
    end if
    if $threshold_{count} \geq threshold_{explore}$ **then** ▷ *stop exploration*
      $DE_{current} \leftarrow DE_{normal}$
      $threashold_{count} \leftarrow 0$
      $immigrants_{current} \leftarrow immigrants_{normal}$
    end if
    for each $\mathbf{x}_i(t) \in n_s$ **do**
      $\mathbf{x}'_i(t) \leftarrow DE_{current}(\mathbf{x}_i(t), n_s)$
    end for
    $immigrants \leftarrow$ ▷ *generate immigrants*
      $\text{GENERATE}(immigrants_{current})$
    $immigrants \leftarrow \text{EVALUATE}(immigrants)$ ▷ *evaluate immigrants*
    $n_s \leftarrow \text{COMBINE}(n_s, immigrants)$ ▷ *diversify population*
    $n_s \leftarrow \text{ITERATEDLOCALSEACH}(n_s, ILS)$ ▷ *local hill-climber*
    if $threshold_{count} < threshold_{explore}$ & $DE_{current} = DE_{explore}$ **then**
      $threshold_{count} \leftarrow threshold_{count} + 1$
    end if
    $t \leftarrow t + 1$
  until stopping condition(s) satisfied

ation of r_{cloud}. Neutral particles move though the search space using PSO equations (3.1) and (3.2) and also consider the quantum particles when calculating their updated velocity vectors. Algorithm 3.7 provides the pseudo-code for the QPSO algorithm.

The size of r_{cloud} restricts the area within which quantum particles may move around the current global best particle. Figure 3.1 illustrates a two-dimensional problem search space and highlights the search space regions that quantum particles will not explore, even if the diameter of the quantum cloud extends to the full extent of the problem domain. The shaded areas in figure 3.1 are never explored by the quantum particles, preventing the discovery of new solutions in these regions. When r_{cloud} is small, the area around the global best particle may not allow detection of new, better optima. On the other hand, when r_{cloud} is large, the area around the global best may be large enough to hide new optima. The value of the r_{cloud} parameter is therefore problem dependent and requires tuning to best optimise the area within which the quantum particles can explore the problem landscape to locate new optima.

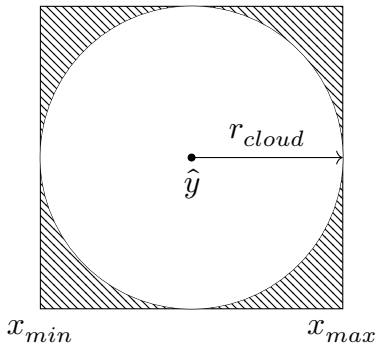


Figure 3.1: r_{cloud} as the extent of the domain

In addition to the r_{cloud} parameter and the control parameters required for the velocity update (defined in equation (3.1)) of the neutral particles, the QPSO also specifies a *split ratio* to subdivide particles into either quantum or neutral particle subgroups. The particle split process occurs at the start of the optimisation process (*i.e.*, at initialisation) only.

3.3 Co-Evolutionary Algorithms

This class of algorithm presents an execution framework approach instead of a single algorithm [175]. Co-evolution implies that multiple candidate solution populations evolve simultaneously, where the co-evolution is driven by either competition [101, 227] or cooperation [88, 98, 149, 163, 175, 235, 256, 284]. Regardless of strategy followed, populations share information with each

Algorithm 3.7 Quantum Particle Swarm Optimisation

$(S_n, S_q) \leftarrow$ Initialise swarm with subgroups for neutral and quantum particles
$r_{cloud} \leftarrow$ Set to user defined value
repeat
  for each neutral particle, $p \in S_n$ **do**
    update velocity with equation (3.1)
    update particle position with equation (3.2)
    calculate and assign objective function value
    update personal best position **iff** position within problem domain
  end for
  for each quantum particle, $p \in S_q$ **do**
    update position with equation (3.6)
    calculate and assign objective function value
  end for
until stopping condition(s) satisfied

other. The shared information is used to determine how the individual populations will then react to other populations. Generally, the framework does not prescribe how the populations interact and learn from each other and only makes provision for the sharing of information between the populations. For example, within the competitive strategy, the best performing candidate solutions from each population compete to determine the best performing solution. The populations of the "losing" candidate solutions can then use the result of the competition to better compete with the winning population in subsequent iterations. Within cooperative strategies, the best population solutions may determine how much they individually contribute to the global problem solution and adjust accordingly. Figure 3.2 provides a simplified illustration of the co-evolutionary process. Within the co-evolutionary framework, any population based algorithm control and manage a population. Thus, the co-evolutionary approach is valid for DOPs by allowing each population to adjust to the changing problem search space.

Based on the general framework above, together with the evolution strategy, different algorithms have been proposed to solve both SOPs and DOPs. Examples of co-evolutionary algorithms for the respective problem types include Cooperative Particle Swarm Optimisation [284] and Cooperative Coevolutionary Differential Evolution with Improved Augmented Lagrangian [98].

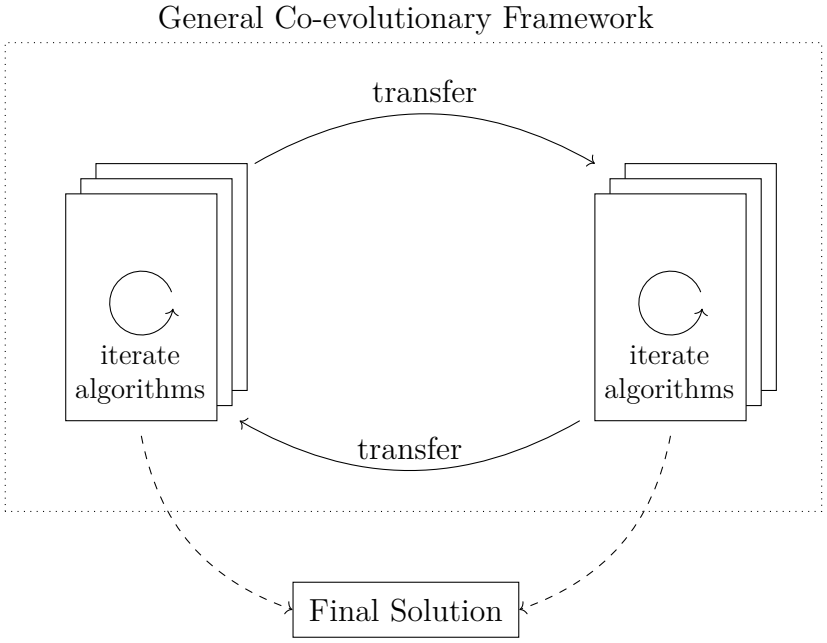


Figure 3.2: General Co-evolutionary Framework

3.4 Constraint Handling and Dynamic Optimisation Algorithms

Chapter 2 introduced the inter-dependency of optimisation algorithm, optimisation problem and problem constraints. The inter-dependency is illustrated in figure 2.1. When constraints are applied to the optimisation problem space, the optimisation problem is divided into feasible and infeasible search space regions. Because infeasible regions contain solutions that may not be considered, the optimisation algorithm should avoid these regions and focus on feasible regions instead. Constraint handling approaches allow for the transformation of constrained optimisation search spaces into unconstrained optimisation search spaces. The transformation represents a new problem landscape where the constrained regions of the problem search space are no longer present. An alternative approach to problem search space transformation places the responsibility of the constraint handling with the optimisation algorithm, leaving the constrained optimisation problem unchanged.

Within dynamic optimisation, the focus of the optimisation algorithm is to track and maintain solutions within the changing search space. The change period (see section 2.1) of the DOP exposes an instance of a static optimisation problem upon which the dynamic optimisation algorithm executes before the next problem landscape change. Consider a dynamic variant of PSO executing on a dynamic search space and how the memory of the particles update after the problem landscape undergoes change. The particle's current and previous best candidate solutions could be in either feasible or infeasible search space

regions within the updated problem landscape. The following scenarios may occur with respect to the current and previous best particle positions after being re-evaluated by the updated objective function:

1. **Feasible current position, feasible best position**: With feasible candidate solutions, no other consideration is needed and the PSO can proceed as if no search space change had occurred.

2. **Feasible current position, infeasible best position**: The particle will update its best position to the next feasible position found after moving through the problem search space in the next iteration of the algorithm.

3. **Infeasible current position, feasible best position**: The movement of the particle using equations (3.1) and (3.2) will hopefully attract the particle towards feasible problem space in subsequent algorithm iterations.

4. **Infeasible current position, infeasible best position**: Both particle candidate solutions are infeasible. The particle itself will remain within infeasible problem space until the social attractor can encourage the particle to move towards other feasible particles.

Importantly, the dynamism of the problem landscape and the subsequent change response from the optimisation algorithm is limited to how the optimisation algorithm adapts to cope with the changing search space. The optimisation problem constraints will determine the feasibility of the candidate solutions within the optimisation algorithm. The feasibility of the candidate solution and memory items will impact the decisions taken by the optimisation algorithm, based on candidate solution feasibility.

Table 2.1 lists constraint handling methods that are predominately related to the optimisation problem, but are managed by the optimisation algorithm. Additionally, these constraint handling strategies do not consider any distinction between static and dynamic optimisation algorithms. Consider the use of the belief space within CE in order to handle problem constraints. By updating the belief space, the optimisation algorithm will continue to propagate any discovered experience of infeasible regions within the optimisation problem in subsequent iterations. At the same time, search space changes may result in outdated belief space knowledge, which is one of the main problems associated with dynamic optimisation algorithms. As a result, reacting to problem search space changes requires belief space knowledge to update any problematic information stored within the belief space. Possible update mechanisms might include the purging of any belief space information that can not be updated satisfactorily. A similar scenario is present within algorithms that manage multiple populations. When the landscape change occurs, all of the

sub-populations should adapt to the change. Ensuring that sub-populations remain consistent with the changed optimisation problem is important not only for algorithm correctness, but also to allow the constraint handling method to behave correctly. Due to the inherent complexity introduced by certain constraint handling approaches, it may be preferable to consider simpler constraint handling approaches [46, 47, 287].

Simpler constraint handling methods are particularly favourable when little information is known about the underlying optimisation problem. Repair strategies, special operators and problem reformulations all benefit from problem landscape information which may not be available. When problem constraints are also dynamically changing, the resulting problem search space becomes more mysterious. The interaction between the constraint and objective spaces may produce feasible and infeasible search space regions that are irregular in shape. Problem landscape effects, such as max-blending, may become more pronounced once the constraint landscape is also considered. It is unknown if sophisticated constraint handling techniques are able to cope with the resulting problem search spaces, nor if any information may be derived from the search space to aid the optimisation algorithm.

3.5 Conclusion

Population-based optimisation algorithms have been discussed in this chapter. Static optimisation algorithms were first introduced to provide an understanding of how population-based optimisation algorithms operate in order to provide solutions to static optimisation problems. When considering the application of population-based optimisation algorithms on dynamically changing problem search spaces, the deficiencies of population-based optimisation algorithms were highlighted.

Dynamic optimisation algorithms are modifications of static optimisation algorithms that allow the same static optimisation to operate within and cope with the changes to dynamic optimisation problems. The strategies to allow for an optimisation algorithm to successfully locate and track changing optima were discussed, together with the intertwined relationship with constraint handling approaches.

Optimisation problem constraints restrict the regions within the optimisation problem where valid solutions may be found. The constraint handling approaches that exist (mentioned in section 2.3.2) are not always solely applied to the optimisation problem and require that the optimisation algorithm also be compliant with the constraint handling method. The addition of algorithm modifications in addition to the dynamism within the optimisation problem increase the overall optimisation complexity, with constraint handling further adding to the complexity of the optimisation process.

Part II

Complexity of Dynamic, Constrained Optimisation Problems

Chapter 4

Dynamic, Constrained Optimisation Problems

> *It is change, continuing change, inevitable change, that is the dominant factor in society today. No sensible decision can be made any longer without taking into account not only the world as it is, but the world as it will be.*
>
> Isaac Asimov

Optimisation problem constraints render parts of the problem search space infeasible. Such infeasible regions describe parts of the search space that an optimisation algorithm should not consider to contain valid solutions to the optimisation problem. Optimisation problem constraints may also change over time and may change independently from the optimisation problem. A static optimisation problem may also be transformed into a dynamic optimisation problem simply by allowing the problem constraints to change over time, changing the feasibility of candidate solutions. As a result, the problem search space for a constrained optimisation problem is determined by considering the combination, or composition, of *both* the objective function and constraint functions.

Nguyen and Yao [207, 208] described a collection of environments which contain such constraints, collectively referring to these environments as DCOPs. DCOP environments may present the following behaviour:

- Both the problem search space and the constraints within the environment are dynamic and change over time.
- The constraints remain static, allowing the problem search space to change over time.
- The problem search space remains unchanging, but the constraints change over time.

Due to the interaction of constraints on the search space over time, feasible regions of the problem search space may become infeasible and vice versa. Nguyen and Yao [207, 208] also discussed the characteristics which are important in order to characterise DCOP problems:

1. constraints may result in changes to the shape, percentage, or structure of feasible and infeasible regions;
2. the global optima may switch from a disconnected feasible region to another in problems with disconnected feasible regions; and
3. environments with a static objective function and changing constraints may expose new, better optima without changing existing optima.

At the time of publication, Nguyen and Yao [208] stated that no benchmark functions existed which fulfil the defined criteria, even though Liu [171] and Richter [245] both provided proposals for benchmark functions that did satisfy the criteria.

This chapter discusses DOP benchmark problems in section 4.1, which is expanded on with the inclusion of problem constraints in section 4.2. Due to the inherent complexity of dynamic, constrained optimisation problems, the resulting landscapes should be examined by sampling different metrics which are discussed in section 4.3. Section 4.4 concludes this chapter.

4.1 Dynamic Optimisation Benchmarks

In order to evaluate the effectiveness of an algorithm on a dynamic environment, an algorithm is first evaluated against an accepted set of benchmark problems. These benchmark problems are generally accepted by the research community as representative of conditions which will test the robustness of an optimisation problem. The evaluation criteria for dynamic optimisation benchmark problems is discussed in section 4.1.1, with the rest of this section discussing the different proposed DOP benchmark problems that are available within literature.

4.1.1 Evaluation Criteria for Dynamic Benchmark Problems

Ursem *et al.* [282] have questioned if benchmark problems for dynamic environments are even appropriate, given that the proposed academic problems are far simpler than the complexity found in real-world dynamic problems. On the other hand, little has been done to classify and understand real-world dynamic problems [26]. However, the purpose of academic benchmarks is not to replicate the complexity of real-world problems. Instead, benchmarks allow for a

simplified recreation of a problem environment (which may be impossible with a real-world problem) and focus on individual aspects that an algorithm should be able to cater for. Building up from a set of simple problems to more complex problems provides a progression for algorithm development. Furthermore, academic benchmarks are important because researchers *still* do not understand all the complexities of dynamically changing environments. Nguyen *et al.* [209] identified the following criteria with which a benchmark may be classified:

- **Time-linkage**: Whether future optimisation problem environments depend on current and/or previous solutions found by an optimisation algorithm.
- **Predictability**: Whether subsequent environment changes follow a pattern, then the observed patterns may present as fixed-step sizes, repeated periodic changes or as predictable change intervals.
- **Visibility**: Whether an optimisation algorithm know when environmental changes occur, and if not, can a minimal number of detectors be used to detect the changes.
- **Constraints**: Whether the optimisation problem space contain constraints and do these constraints change over time.
- **Number of objectives**: Whether more than one objective function is present.
- **Factors that change**: Aspects of the problem environment that change, including the problem domain, objective functions, or other parameters.

4.1.2 DF1 Problem Generator

Morrison and De Jong [199] defined a process whereby a DOP may be generated based on a set of parameters and a static basis function:

$$f(\mathbf{x}) = \max_{i=1,\dots,N} \left[H_i - R_i * \sqrt{\Sigma_{i=0}^{n}(x_i - X_i)^2} \right] \tag{4.1}$$
$$H_i \in [H_{base}, H_{base} + H_{range}]$$
$$R_i \in [R_{base}, R_{base} + R_{range}]$$
$$X_i \in [-1, 1]$$

where H_i is the height range for peak i, R_i is the slope coefficient and n is the problem dimension. The subscripts *base* and *range* respectively refer to the absolute minimum and variable ranges for the peak height and slope.

By sampling a Gaussian distribution, variations can be introduced for each peak within the optimisation problem landscape. The definition from the literature embeds the sampling process within the value selection for H_i

and R_i. Although the DF1 generator is able to produce different dynamic environments through the provided set of generator parameters, the result is a set of peaks that are all fairly similar.

The landscape generator did, however, demonstrate that a generator for dynamic environments is a feasible approach to obtain a diverse set of problem landscapes. Usage of the max function also allowed for the creation of environments where the number of optima could change based on the effect of *max blending*. Max blending describes the process whereby a peak within the problem environment seems to disappear as it moves underneath a larger peak, and equation (4.1) only considers the maximum value for a given candidate solution.

4.1.3 Moving Peaks Benchmark

The MPB [23–25] is a function generator that produces dynamic optimisation problems based on a set of input parameters. The problem instances produced by the function generator contain several independent peaks within a multi-dimensional problem landscape. The MPB could be seen as the natural evolution of the DF1 generator, but with more precise control over the generated problem landscapes. Each peak maintains information about the height, width and position within the optimisation problem search space. The quality of candidate solutions is quantified by taking the maximum of all evaluated peaks within the problem instance:

$$F(\mathbf{x}, t) = \max\{B, p_0(\mathbf{x}, e_o), p_1(\mathbf{x}, e_1), \dots, p_n(\mathbf{x}, e_n)\} \tag{4.2}$$

where $p_0, p_1, \dots, p_n$ are individual peak functions defined by the set of peak parameters e_i, which evaluate the candidate solution, $\mathbf{x}$, for time-step t. The value of B in equation (4.2) defines the basis function landscape which has a default value of 0 because the MPB is a maximisation problem.

The initial parameter values for the peak properties are generated by sampling a probability distribution. The peak environment parameters, e_i, define a record-like structure that maintains the properties for a peak and includes:

- *minHeight* and *maxHeight* properties which define the upper and lower bound values for the height of the peak;
- *minWidth* and *maxWidth* provide the upper and lower bound values for the width of a peak;
- Problem domain which determines the bounds within which a peak may be located;
- Peak location within the problem search space, $\mathbf{v}$, together with the current height, h, and width, w;

- *shift vector*, $\mathbf{s}_v$, with the same dimensionality as the peak position vector. The shift vector influences the movement of the peak during an environment change.

Each independent peak within the problem search space is evaluated using the peak environment parameters as follows:

$$p_i(\mathbf{x}, \{\mathbf{v}, h, w\}) = h - w\sqrt{\sum(\mathbf{x} \odot \mathbf{v})_i} \tag{4.3}$$

where $\odot$ is a binary function which determines the component-wise squared difference between the problem search space position $\mathbf{x}$ and the position of the peak $\mathbf{v}$ within the problem search space.

The MPB generator defines a recurrence relation whereby updated peak environment parameters are constructed using the current set of peak environment parameters. The updated peak environments describe properties for a new set of peaks which have moved within the problem bounds and define the changed optimisation problem search space. The peak environment modification necessitates that severity thresholds are used to control the amount of variation applied to the peak parameters in order to construct the changed problem landscape:

- *hSeverity* and *wSeverity* determine the scaling factors for peak height and width adjustments;
- The *change severity*, s, is a constant influencing the amount of peak change between subsequent landscapes;
- $\sigma(t) \sim N(0, 1)$; and
- A coefficient λ to scale the amount of random peak movement.

The previously mentioned recurrence relation which updates the peak environment parameters for the next time step is formally defined as:

$$\begin{aligned} e_i(t+1) = \{e_i(t) \mid\ & h = e_i(t)\{h\} + hSeverity \times \sigma(t), \\ & w = e_i(t)\{w\} + wSeverity \times \sigma(t), \\ & \mathbf{s}_v = \frac{s}{\parallel \mathbf{p}_r + e_i(t)\{\mathbf{s}_v\} \parallel}((1-\lambda)\mathbf{p}_r + \lambda e_i(t)\{\mathbf{s}_v\}) \\ \} & \end{aligned} \tag{4.4}$$

where $\mathbf{p}_r$ is a random vector normalised to length s for the current time step t.

Adjustment of the input parameters provided to the MPB function generator allows for the representation of all scenarios within the classification of Duhain and Engelbrecht [77], thereby subsuming the classifications of Angeline [3], De Jong [62], Eberhart and Shi [81], and Hu and Eberhart [123]. The subsections that follow discuss the MPB parameter selections used within the more complete classification of Duhain and Engelbrecht [77].

4.1.3.1 Peak Movement

The peak movement classification of Angeline [3] is expressed in the MPB input parameters as:

- Linear peak movement: $\lambda = 1.0$ and $s \neq 0$
- Circular peak movement: $s = 0$ and a rotation matrix is applied to equation (4.3)
- Random peak movement: $\lambda = 0.0$ and $s \neq 0$

4.1.3.2 Optima movement

The problem space classification of Eberhart and Shi [81] and Hu and Eberhart [123] requires the following MPB input parameters:

- Type I: $hSeverity = 0$ and $s \neq 0$
- Type II: $hSeverity \neq 0$ and $s = 0$
- Type III: $hSeverity \neq 0$ and $s \neq 0$

4.1.3.3 Spatial and Temporal Environment Changes

Environment categories based on spatial and temporal changes are achieved with the following parameters:

- Progressive environments: s, $hSeverity$ and $wSeverity$ are all set to low values in relation to the size of the problem search space. Environment changes are applied frequently to generate new problem environments.
- Abrupt environments: at least one of the parameters controlling the spatial changes (s, $hSeverity$, $wSeverity$) are set to a high value based on the search space. Environment changes are applied with at a low frequency.
- Chaotic environments: high spatial as well as high temporal change values are defined to create almost random new problem environments. Additionally, the frequency of environment change is high.

To differentiate the features of the problem instances in the classification of Duhain and Engelbrecht [77], the problem instances are named according to the modification properties present within the dynamic optimisation problem. Problem instance names are a three-letter acronym as mentioned in section 2.2.5. Consider the problem instance acronym `A2C`. This acronym defines the abruptly changing, type II, circular environment. The notion of the acronym may be expanded in order to refer to the collective behaviour of a

group of problem instances. An asterisk (*) may be substituted in for one of the acronym characters thereby creating a category of problem instances that adhere to the non-substituted environment properties. For example, the acronym *2L groups all problem instances that are type II problem environments with linear peak movement.

4.1.3.4 Peak Movement Characteristics

If temporal changes of the problem instances are temporarily ignored, the movements of the individual peaks within the MPB may be described:

- Random peak movement: The individual peaks within the problem landscape undergo stochastic changes to their height, width and search space location as shown in equation (4.4). The same process to transform the peaks is used for the different peak modification types with the environment parameters negating certain changes to the peak environment. For example, the peak environment parameters for type II environments prevent any changes to the position of a peak within equation (4.4) and randomly adjust the height and/or width of the peak instead. Type I environments adjust the position of the peak and type III environments apply modifications to peak position, height and/or width.

- Linear peak movement: Individual peaks are adjusted from an initial starting point and are moved through the environment linearly. Type I environments experience peaks that move from one spectrum of the n-dimensional problem space to the other and then back again. Type II environments linearly adjust the height of the peak based on a random starting point and direction (up or down) without moving the peak. Type III environments combine both type I and type II behaviour, as expected.

- Circular peak movement: Peaks within the problem landscape are moved in a circular pattern. Type I and type III environments use a landscape rotation [300] to move peaks in a circular fashion, with the rotation centred at the central point of the domain in all n-dimensions of the problem landscape. The rotation of the peak is along a randomly selected plane, assigned at problem instance creation, which does not change. The rotation is along a single hyper-plane to ensure that a rotation angle of $\theta = 2\pi$ will once again result in the original problem landscape. Rotating more than a single hyper-plane at once will violate this property and will invalidate the entire rotation process. However, it is possible to compose different plane rotations into a single rotation by applying rotations in different hyper-planes sequentially. The composed rotation is, however, *not* commutative and the order of rotations from left

to right is important. If the rotation ordering is changed, the resulting rotation matrix also changes.

For the rotation on a hyper-plane, the rotation matrix $R_{x_1x_2}(\theta)$ rotates the hyper-plane x_1x_2 by θ degrees. A rotation matrix in the plane x_1x_2 in n-dimensions has the form:

$$\begin{array}{c} \\ x_1 \\ x_2 \\ x_3 \\ x_4 \\ \vdots \\ x_n \end{array} \begin{array}{c} \begin{array}{cccccc} x_1 & x_2 & x_3 & x_4 & ... & x_n \end{array} \\ \begin{bmatrix} cos(\theta) & -sin(\theta) & 0 & 0 & ... & 0 \\ sin(\theta) & cos(\theta) & 0 & 0 & ... & 0 \\ 0 & 0 & 1 & 0 & ... & 0 \\ 0 & 0 & 0 & 1 & ... & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & 0 & ... & 1 \end{bmatrix} \end{array} \tag{4.5}$$

The rotation of peaks does not use the defined shift vector. A cycle length C defines the degree of peak movement such that the circular movement mimics the movement that would occur if the shift vector and severity parameters were used. For example, given a continuous domain of $[0, 100]^d$, a rotation cycle length of $C = 314$ ensures that all points with a radius of $r = 50$ from the midpoint of the domain $(50, 50)^d$ change position by $\frac{2\pi r}{314} \approx 1$. A cycle length of 314 is close to $s = 1$ for progressive environments. Similarly, a cycle length $C = 62$ results in $\frac{2\pi r}{105} \approx 3$ which is approximately $s = 3$ for abrupt and chaotic environments. Consequently, using these cycle lengths produces peak movements in a manner similar to the movements produced when using the associated s values.

Type II environments produce a circular peak movement by allowing the height of the stationary peeks to oscillate between the peak environment's *minHeight* and *maxHeight* values. Upon reaching the limit of the defined range, the direction of the peak height movement changes to allow the peak height to either grow or shrink to provide the circular movement pattern.

4.1.4 Generalised Dynamic Benchmark Generator

Li and Yang [160] proposed the generalized dynamic benchmark generator (GDBG) that allows for the generation of problem landscapes within binary-valued spaces, combinatorial spaces and real-valued (or continuous) landscapes. The definition for the generalised generator function is as follows:

$$F = f(\mathbf{x}, \phi, t) \tag{4.6}$$

where F is the optimisation problem with the cost function f, $\mathbf{x}$ is the candidate solution for the optimisation problem for the current time interval, t. The set

of system control parameters, ϕ, is updated for the subsequent environment, using a change strategy procedure referred to as *DynamicChanges* and is symbolically represented as $\Delta\phi$. The update process on the next time interval $t+1$ is described as:

$$f(\mathbf{x}, \phi, t+1) = f(\mathbf{x}, \phi(t) \oplus \Delta\phi, t) \tag{4.7}$$

The framework of $\Delta\phi$ changes is a system of six possible change types:

1. **Small step**: $\Delta\phi = \alpha \cdot \|\phi\| \cdot r \cdot \phi_{\text{severity}}$
2. **Large step**: $\Delta\phi = \|\phi\| \cdot (\alpha \cdot sign(r) + (\alpha_{max} - \alpha) \cdot r) \cdot \phi_{\text{severity}}$
3. **Random**: $\Delta\phi = N(0,1) \cdot \phi_{\text{severity}}$
4. **Chaotic**: $\phi(t+1) = A \cdot \phi(t) \cdot (1 - \phi(t)/\|\phi\|)$
5. **Recurrent**: $\phi(t+1) = \phi_{min} + \|\phi\|(\sin(\frac{2\pi}{P}t + \varphi) + 1)/2$
6. **Recurrent with noisy**:
   $\phi(t+1) = \phi_{min} + \|\phi\|(\sin\frac{2\pi}{P}t + 1)/2 + N(0,1) \cdot noisy_{severity}$

Within the above framework, $\|\phi\|$ is the change in the range of ϕ, with $\phi_{severity} \in (0,1)$ and $noisy_{severity} \in (0,1)$. The values α and α_{max} are constants in the range $(0,1)$ and P is the period for both recurrent change and recurrent change with noise. The value ϕ defines the initial phase, whilst r is a random number in $(-1,1)$. $N(0,1)$ is a normal distribution with a mean of 0 and standard deviation of 1. The function, $sign(x)$, returns values of 1 when $x > 0$, -1 when $x < 0$ and 0 otherwise.

For the generation of continuous-valued problem landscapes, the GDBG defines a rotation dynamic benchmark generator (RDBG) which incorporates a system of control parameters which are updated using the framework of $\Delta\phi$. The problem generator is formulated by $\phi = (\vec{H}, \vec{W}, \vec{X})$ where $\vec{H}, \vec{W}$ and $\vec{X}$ respectively denote the height, width and position of a peak within the generated problem landscape. The cost function within RDBG is defined as:

$$f(x, \phi, t) = \min_{i=1}^{m}(H_i(t) + W_i(t) \cdot (\exp\left(\sqrt{\sum_{j=1}^{n}\frac{(x_j - X_j^i(t))^2}{n}}\right) - 1)) \tag{4.8}$$

where m is the number of peaks with a dimension of n. The peak width and height are adjusted as follows:

$$\vec{H}(t+1) = DynamicChanges(\vec{H}(t)) \tag{4.9}$$

$$\vec{W}(t+1) = DynamicChanges(\vec{W}(t)) \tag{4.10}$$

Unlike the peak shifting present within the MPB generator, the RDBG moves peaks within the problem landscape though the use of a rotation matrix. The rotation matrix is derived using the procedure outlined by Weicker and Weicker [300] which was described within the MPB circular peak rotation procedure within section 4.1.3.4.

4.1.5 Compositional function generator

Liang *et al.* [165] proposed an approach to generate dynamic environments through the composition of static environment basis functions. The static basis functions are predominately selected from the well-established set of static environment benchmarks that are commonly used within literature. Such functions include the Spherical, Rastrigin, Griewank, Ackley and Schwefel functions, amongst others. The compositional approach for the dynamic composition benchmark generator (DCBG) is achieved through the use of the composition function:

$$F(\mathbf{x}, \phi, t) = \sum_{i=1}^{m} \left(w_i \cdot \left(f_i' \left(\left(\mathbf{x} - \vec{O}_i(t) + O_{iold} \right) / \lambda_i \cdot \vec{M}_i \right) + \vec{H}_i(t) \right) \right) \quad (4.11)$$

where $\phi = (\vec{O}_i, \vec{M}_i, \vec{H}_i)$ is the system control parameter for the generator, $f_i(\mathbf{x})$ is the i-th composition function (of m functions) used to construct the composition function $F(x)$. The function $f_i'(\mathbf{x}) = C \cdot f_i(\mathbf{x}) / |f_{max}^i|$ scales the contribution of each $f_i(\mathbf{x})$ relative to the maximum value, $f_{max}^i = f_i(\mathbf{x}_{max} \cdot M_i)$. $\vec{M}_i(t)$ is the orthogonal rotation matrix for each $f_i(\mathbf{x})$, calculated upfront and thereafter remains unchanged. $\vec{O}_i(t)$ defines the optimum of the changed $f_i(\mathbf{x})$ as a result of the rotation in time-step t. $O_{i_{old}}$ is the optimum of the original $f_i(\mathbf{x})$ without any applied changes.

The weight w_i applied to each of the $f_i(\mathbf{x})$ functions within equation (4.11) is calculated as:

$$w_{ia} = \exp\left(-\operatorname{sqrt}\left(\frac{\sum_{k=1}^{n} \left(x_k - o_i^k + o_{i_{old}}^k \right)^2}{2n\sigma_i^2} \right) \right) \quad (4.12)$$

$$w_{ib} = \begin{cases} lw_{ia} & \text{if } w_{ia} = \max(w_{ia}) \\ w_{ia} \cdot (1 - \max(w_{ia})^{10}) & \text{if } w_{ia} \neq \max(w_{ia}) \end{cases} \quad (4.13)$$

$$w_i = w_{ib} / \sum_{i=1}^{m} w_{ib} \quad (4.14)$$

where σ_i is the coverage range factor of $f_i(\mathbf{x})$ with a default value of 1.0; λ_i is the stretch factor for each $f_i(\mathbf{x})$, defined as

$$\lambda_i = \sigma_i \cdot \frac{X_{max} - X_{min}}{x_{max}^i - x_{min}^i} \quad (4.15)$$

where $[X_{min}, X_{max}]^n$ is the problem domain for $F(\mathbf{x})$, whereas $[x_{min}^i, x_{max}^i]^n$ is the search domain for $f_i(\mathbf{x})$. The system parameters for $\vec{H}$ and $\vec{O}$ are adjusted as follows:

$$\vec{H}(t+1) = DynamicChanges(\vec{H}(t)) \quad (4.16)$$

$$\vec{O}(t+1) = DynamicChanges(\vec{O}(t)) \quad (4.17)$$

The values $\vec{H}$ and $\vec{O}$ are the same as $\vec{H}$ and $\vec{X}$ within RDBG and are adjusted in the same way using the change system, $\Delta\phi$.

The implementation of the DCBG benchmark may prove to be challenging due to the required variable state management for the additional control parameters within the generator function. Importantly, the optimum value for each of the constituent composition functions must be known ahead of time in order to apply the compositional generator. As a result, the DCBG may produce an environment which is computationally expensive to construct and/or evaluate, and to update upon an environment change.

4.1.6 Free Peaks Generator

Li [159] proposed a new landscape generator based on the idea that the max blending effect within the MPB and DF1 is incorrect. The proposed generator firstly divides the problem search space bounds up into smaller sub-spaces using a k-d tree [11]. Following the subdivision of the problem space, a function is associated with each subspace in order to define the landscape for the subspace region. The branching nodes of the k-d tree represent the points within the problem search space where a hyperplane divides the search space into two sub-spaces. The leaf nodes of the k-d tree represent the final sub-spaces which will have peak functions assigned to them in order to construct the problem search space landscape.

Eight individual landscape functions are given in [159], but any number of landscape functions may be provided based on the landscape requirements. The generator framework also allows for changes to the landscape to occur over time which include the peak location, peak height, the peak basin of attraction (based on the subspace division) and the number of peaks in the environment.

The free peaks approach provides a piece-wise approach to the construction of the problem landscape, but at the same time the procedure does not define a clear evolution process for the environment over time. For example, the number of peaks may change after an environment change, requiring that the problem space be subdivided once again in order to reduce or increase the number of available peaks. Such changes can be seen as a drastic, chaotic change within the problem environment which may not be desired behaviour for the dynamic environment. It is also unclear how the benefits of the generator listed within [159] are outright better than the other generator based landscape generation approaches. The landscapes produced by the generator have not been categorised in order to define the landscape behaviour characteristics for the set of generator configuration parameters and environment changes.

4.2 Constrained Dynamic Optimisation Benchmark Problems

DODC problems are the most challenging kind of optimisation problem to obtain solutions for. This is predominately due to the complexity introduced by the problem definition and that problem changes affect the optimisation algorithm. The DCOP are a class of problem that is not fully understood, with a sparse selection of benchmark problems available. In the sub-sections that follow, the current set of available benchmark problems are discussed together with the current limitations. Section 4.2.1 discusses the sets of benchmark problems used in the IEEE Congress on Evolutionary Computation (CEC) competitions, with section 4.2.2 discussing the G24 benchmark problems.

4.2.1 CEC Competition Benchmarks

The CEC has hosted multiple competitions in order to encourage the development of optimisation algorithms for constrained optimisation problems.

The CEC2006 conference competition [167] focused on single objective, static optimisation problems which contain static problem constraints. The organisers of the competition also provided the target solution for each of the problem instances, enabling comparisons between the different participating algorithms. All defined problem instances were limited to minimisation problems. The benchmark problems consists of 24 problem instances (`g1-g24`) that are either linear, non-linear, quadratic, cubic or polynomial functions in multiple dimensions. Problem constraints consist of equality and/or inequality constraint functions which are specified for each problem instance. The benchmark problems were prepared exclusively for the competition, but have been used in studies since the CEC2006 competition, especially when comparing against an algorithm which took part in the competition.

For the CEC2010 competition on constrained real-parameter optimisation [182], a total of 18 unique constrained problem instances were described within the technical report. The focus of the benchmark problems within the CEC2010 competition was on constrained problems in higher dimensions. This is in contrast to the 2006 competition where the focus was on static optimisation problems. The number of constraints for each of the benchmarks varied from 0 to 3 constraints, where the constraints themselves are either separable or non-separable functions. As with the CEC2006 competition, the benchmark problems are specific to the CEC2010 competition.

A new set of benchmark problems was proposed in the technical report for the CEC2017 competition [306]. The intention of this new set of benchmark problems was to address the concern that the benchmark problems from the CEC2006 and CEC2010 technical reports no longer provide a challenge because they had been solved successfully. The new benchmark problems consisted of

28 constrained problems with dimensions of 10, 30, 50 and 100. These new benchmark problems aimed to be scaling functions that would compete with other problems within the Big Data industry where problems contain hundreds of decision variables together with a wide variety of problem constraints. As a result, the new benchmark problems contain a wider range of different constraint functions when compared to the benchmark problems from previous competitions.

4.2.2 G24 Benchmark Functions

Nguyen and Yao [207] proposed a set benchmark problems, referred to as the G24 benchmark (not to be confused with the function names within the CEC2006 competition benchmark). A benchmark problem generator was proposed to generate search space landscapes that change over time as well as static problem landscapes. In addition, the generator allowed for the inclusion of static constraints into the generated landscape, but also allowed the constraints themselves to change over time. The result is a problem instance generator for DCOPs which aims to fulfil the requirements of desirable optimisation problems for dynamic environments, discussed in section 4.1.1.

The problem generator is based on a mathematical framework [210], from which the problem instances are derived. Generalisation of the mathematical framework into its dynamic version requires that the parameters of the static framework become time-dependent parameters. In other words, for the set of problem parameters P, each parameter $p_i \in P$ may be generalised into a time-dependent parameter version by replacing p_i with $p_i(t)$. G24 benchmark problem instances are based on a basis static function [92] defined as:

$$f(\mathbf{x}, t) = -(X_1(x_1, t) + X_2(x_2, t)) \tag{4.18}$$

subject to:

$$\begin{aligned} g_1(\mathbf{x}, t) = &-2Y_1(x_1, t)^4 + 8Y_1(x_1, t)^3 - 8Y_1(x_1, t)^2 \\ &+ Y_2(x_2, t) - 2 \\ g_2(\mathbf{x}, t) = &-4Y_1(x_1, t)^4 + 32Y_1(x_1, t)^3 - 88Y_1(x_1, t)^2 \\ &+ 96Y_1(x_1, t) + Y_2(x_2, t) - 36 \end{aligned}$$

such that

$$\begin{aligned} X_i(x_i, t) &= p_i(t)(x_i + q_i(t)) \\ Y_i(x_i, t) &= r_i(t)(x_i + s_i(t)) \\ 0 \leq x_1 \leq 3; &\quad 0 \leq x_2 \leq 4 \\ g_1(\mathbf{x}, t) \leq 0; &\quad g_2(\mathbf{x}, t) \leq 0 \end{aligned}$$

where $p_i(t), q_i(t), r_i(t)$ and $s_i(t)$ (for $i = 1, 2$) are the time-dependent parameters. Values for the time-dependent parameters are provided in table 4.1 for each of the problem instance variations.

Although the problem instances within the G24 benchmark problems do address the defined criteria, the benchmark problem instances are limited to a two-dimensional problem domain which reduces the usefulness of the generator. The G24 does, however, indicate that the generator-based approach may be a valid alternative to define different search space environments which can serve as benchmark problems for DCOP instances. Furthermore, basing a DCOP generator on an already established and well understood DCOP problem generator is very advantageous. Allowing such a generator to additionally define constraints on the produced problem will also allow for a more focused understanding of algorithm behaviour, when adding constraints to a DOP instance.

4.3 Landscape Analysis

As the complexity of the problem search space increases, it becomes more difficult to understand the problem. The interaction between constraints and the problem search space is already complicated in two dimensions and scaling to larger dimensions only compounds the complexity further. It is, therefore, advantageous to attempt to understand the problem search space using a set of metrics, based on samples of search space data. From such measurements, details of the problem landscape may be determined before an optimisation algorithm attempts to find solutions through fitness landscape analysis (FLA) [233]. Section 4.3.1 discusses the process of obtaining the sample landscape data, whilst section 4.3.2 describes metrics available to aid in understanding the problem landscape.

4.3.1 Landscape Walks

Landscape walks [177, 181] through the problem landscape define a process to collect an ordered set of multi-dimensional candidate solutions from an optimisation problem. Candidate solutions are sampled at regular intervals across the entire problem domain in order to create the set of points. The resulting ordered set of candidate solutions represents a collection of neighbouring solutions that constitute the walk through the problem search space. Figure 4.1 illustrates neighbouring of candidate solutions within a landscape walk, where the neighbouring candidate solutions are connected with lines. The walk creation process should also adhere to the following guidelines [177]:

- Neighbouring solutions define a loose notion of neighbourhoods based on an arbitrary definition, such as the Euclidean distance between candidate solutions.

Table 4.1: Parameter combinations for the G24 benchmark generator

Problem	Parameters
G24_0	$p_1(t) = \sin\left(k\pi t + \frac{\pi}{2}\right); \quad p_2(t) = 1$
G24_1	$p_1(t) = \sin\left(k\pi t + \frac{\pi}{2}\right)$ $p_2(t) = r_i(t) = 1; \quad q_i(t) = s_i(t) = 0$ $i = 1, 2; \quad 0 < k \leq 2$
G24_2	$p_1(t) = \sin\left(\frac{k\pi t}{2} + \frac{\pi}{2}\right)$ $p_2(t) = \begin{cases} t \bmod 2 \neq 0 & \sin\left(\frac{k\pi(t-1)}{2} + \frac{\pi}{2}\right) \\ t \bmod 2 = 0 & \begin{cases} p_2(t-1) & \text{if } t > 0 \\ p_2(0) = 0 & \text{if } t = 0 \end{cases} \end{cases}$ $q_i(t) = s_i(t) = 0; \quad r_i(t) = 1; \quad i = 1, 2$
G24_3	$p_i(t) = r_i(t) = 1; \quad q_i(t) = s_1(t) = 0; \quad i = 1, 2$ $s_2(t) = 2 + t \cdot \frac{x_{2\max} - x_{2\min}}{S}$
G24_4	$p_1(t) = r_i(t) = 1; \quad q_i(t) = s_1(t) = 0; \quad i = 1, 2$ $p_2(t) = \sin\left(k\pi t + \frac{\pi}{2}\right); \quad 0 < k \leq 2$ $s_2(t) = t \cdot \frac{x_{2\max} - x_{1\min}}{S}$
G24_5	$p_1(t) = \sin\left(\frac{k\pi t}{2} + \frac{\pi}{2}\right)$ $p_2(t) = \begin{cases} t \bmod 2 \neq 0 & \sin\left(\frac{k\pi(t-1)}{2} + \frac{\pi}{2}\right) \\ t \bmod 2 = 0 & \begin{cases} p_2(t-1) & \text{if } t > 0 \\ p_2(0) = 0 & \text{if } t = 0 \end{cases} \end{cases}$ $qi(t) = s1(t) = 0; ri(t) = 1; i = 1; 2$ $s_2(t) = 2 + t \cdot \frac{x_{2\max} - x_{2\min}}{S}$
k	Severity of function change Small: $k \sim 0.25$, Medium: $k \sim 0.5$, Large: $k \sim 1$
S	Severity of constraint changes Small: $S = 50$, Medium: $S = 20$, Large: $S = 10$

- The candidate solution collection process must be unbiased and not guided in any way; this is unlike the general process within an optimisation algorithm.
- The length of a walk should also be guided by the computational constraints, which includes the cost of the computation and the total budget available to explore the landscape. If a study aims to explore the behaviour of an optimisation algorithm, a larger computational budget should be allocated to the optimisation algorithm instead of to the walk creation process. If the generated walks through the landscape perform the majority of the candidate solution evaluations, the walk itself is acting as a search for an optimal solution instead of probing the landscape to understand the present characteristics.

Walks may be generated with random sampling or by directing the sampling process across the landscape.

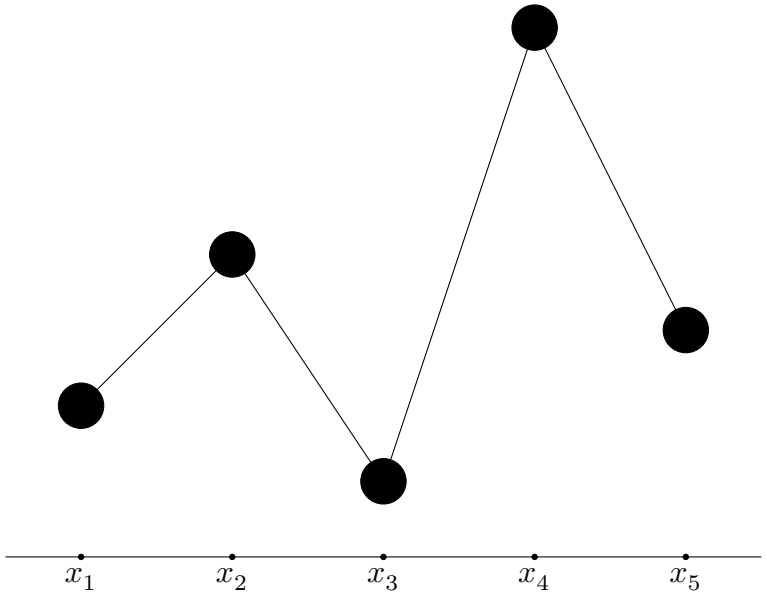


Figure 4.1: Simplified view of neighbouring candidate solutions within a walk. Neighbouring candidate solutions are connected by lines.

4.3.1.1 Simple Random Walk

A simple random walk [290] of the environment landscape can be defined as the process of collecting search space candidate solutions that total the user defined quantity *numSteps*. The initial point for the collection procedure is randomly generated for each dimension within the range $[x_{\min}, x_{max}]$, where $x_{\min}$ and $x_{\max}$ respectively define the lower and upper bounds of the landscape domain.

Each subsequent candidate solution within the walk is based on the most recently collected candidate solution. Given a user-defined step size, s, a "step size vector" is created by sampling a random value between $[-s, s)$ for each dimension within the candidate solution. The step size vector and the most recent current candidate solution within the walk are then combined through a component-wise addition of dimension values in order to produce the next candidate solution of the walk. This process is iterated until *numSteps* candidate solutions are collected, producing the completed walk.

Although the above simple sampling procedure is simple to execute, the results from the walk creation are not desirable. Malan [177] highlighted that the walk creation procedure produces walks that have a tendency to be clustered around limited areas within the search space. It was also observed that the localisation of sampled candidate solutions became more pronounced as the size of s decreased. Furthermore, as *numSteps* reduces, so too does the probability of producing a candidate solution within a different part of the landscape search space.

4.3.1.2 Progressive Random Walk

In order to address the limitations of the simple random walk, Malan [177] proposed an alternative walk generation procedure. The proposed walk generation process would always begin at an edge of the landscape search space, but included a bias to move towards the other end of the landscape search space. The candidate solution selection process remains the same as in the simple random walk above, whereby each dimension within the step vector is sampled randomly within the range $[-s, s)$.

Upon reaching or passing over the opposite landscape boundary, the bias would swap to direct the walk sampling to the starting landscape boundary. The repeating back and forth sampling across the search landscape would continue until *numSteps* candidate solutions have been collected. As a result of the improved spread of candidate solutions across the landscape search space when sampling with the progressive random walk procedure, the procedure is preferred to the simple random walk.

4.3.1.3 Manhattan Progressive Random Walk

An alternative walk generation procedure may be considered when the metric computation process becomes too computationally expensive [148, 177]. For example, metrics that use the Euclidean distance between candidate solutions may require the majority of a computational budget. The Manhattan progressive random walk allows for a simplified walk generation procedure in which a single dimension of a candidate solution is modified in order to produce a new candidate solution. The dimension selected for modification is selected at random and the bias of the progressive random walk to drive the walk across the

problem landscape is still present within the walk creation procedure. Importantly, the size of the modification applied to the randomly selected dimension is a constant size instead of being randomly sampled from the range $[-s, s)$.

4.3.2 Landscape Metrics

Fitness landscape metrics use the resulting walk information in order to quantify the characteristics of a given optimisation problem search space. Assuming that the walk information provides a sufficient number of sampled candidate solutions, as well as the phenotypic information for each point, the fitness landscape metrics may provide insight to the characteristics of the problem search space. The sampled candidate solutions contained within the walk should ideally be from across the entire problem domain, whilst the number of candidate solutions should be as large as possible.

It is recommended to perform a number of independent walks throughout the problem search space in order to determine statistical relevance for the fitness landscape metrics. A number of different fitness landscape metrics are available with a sample of the available metrics discussed in the subsections that follow.

4.3.2.1 Feasibility Ratio

The **feasibility ratio (FSR)** [179] is an approximation of the feasible portion of the problem search space when compared against the total available search space. The metric is simply

$$\text{FSR} = \frac{n_f}{n} \tag{4.19}$$

where n_f is the number of feasible solutions contained within a random walk of the search space.

4.3.2.2 Ratio Feasibility Boundary Crossings

The **ratio feasibility boundary crossings (RFBx)** [179] determines how disjoint the feasible regions of the problem space are. The metric traverses the candidate solutions within a random walk and counts the proportion of neighbouring solutions that cross between feasible and infeasible spaces within the problem search space. If the random walk were to be transformed into a binary string b, the bit value 0 indicates that the solution is within feasible space. Conversely, a bit value of 1 represents a candidate solution within infeasible space.
The RFBx metric is calculated as:

$$\text{RFBx} = \frac{\Sigma_{i=1}^{n-1} cross(i)}{n-1} \tag{4.20}$$

where

$$cross(i) = \begin{cases} 0 & \text{if } b_i = b_{i-1} \\ 1 & \text{otherwise} \end{cases} \tag{4.21}$$

4.3.2.3 Dispersion

The **dispersion** [174] metric estimates the global topology of a problem search space from the provided solution information within the random walk. The *dispersion* for a random walk sample determines how spread out the points are in relation to each other, based on the average pair-wise distance between the points. The function of dispersion is based on how the dispersion of a sample of points differs from a subset of better points [174]. The function may either be regarded as *high* or *low*, where the dispersion of the sample of points is respectively higher or lower than the dispersion of the subset of better points. The metric was improved upon in [177] where the candidate solutions were normalised in order to allow for the comparison of dispersion metric values where problems have different domains.

4.3.2.4 First Entropic Measure

The **first entropic measure (FEM)** [177, 288–290] determines the classification of the problem search space as either rugged, smooth or neutral, using the search space information between neighbouring solutions. The result of the measure is a value within the range $[0.0, 1.0]$, which indicates a search space from smooth to total ruggedness.

Although the result of the measure is a single scalar value, the calculation itself is a complex sequence of steps where the change in objective function value between landscape walk points are considered. The value comparison forms a cycle with the last point in the sample being compared against the first sample point within discrete problem spaces. Continuous problem spaces do not close the cycle by comparing the first and last samples. The metric is defined as:

$$\text{FEM} = \max_{\forall \epsilon \in [0, \varepsilon^*]} \{H(\varepsilon)\} \tag{4.22}$$

where $H(\varepsilon)$ is an instance of *b-ary* entropy [247] which determines the entropy between two unique sample points.

4.3.2.5 Fitness Cloud Index

The **fitness cloud index (FCI)** [178] indicates the evolvability of an evolutionary search. The metric makes use of a PSO algorithm to determine the neighbourhoods of a sample of solutions, and normalises all fitness values of the sample considering only solutions that are within the bounds of the problem search space. The resulting value is within the range $[0, 1]$, where 0 indicates

the worst searchability and 1 the best searchability. The measure is defined as:

$$\text{FCI} = \frac{\sum_{i=1}^{n_v} g(\mathbf{x}_i)}{n_v} \tag{4.23}$$

where $\mathbf{x}_i$ is a sampled point from the optimisation problem search space (*i.e.*, $\mathbf{x}_i \in \mathcal{S}$) and n_v is the number of valid points within the subset $\mathcal{S}_v$ such that $\mathcal{S}_v \subseteq \mathcal{S}$, and

$$g(\mathbf{x}_i) = \begin{cases} 1 & \text{if } f(\mathbf{x}_i') < f(\mathbf{x}_i) \\ 0 & \text{otherwise} \end{cases} \tag{4.24}$$

with $\mathbf{x}_i'$ being the associated neighbouring points for the point $\mathbf{x}_i$.

4.3.2.6 Fitness Distance Correlation

The **fitness distance correlation (FDC)** [178] metric informs whether the information presented by a problem space could guide an optimisation algorithm to an optimum. The premise is that a problem is simple to search if the objective function value of solutions increase or decrease as the distance to an optimum increases or decreases. The correlation value may range from -1.0 (totally uncorrelated) to 1.0 (totally correlated) and is calculated as:

$$\text{FDC} = \frac{\sum_{i=1}^{n}(f_i - \bar{f})(d_i^* - \bar{d}^*)}{\sqrt{\sum_{i=1}^{n}(f_i - \bar{f})^2} \cdot \sqrt{\sum_{i=1}^{n}(d_i^* - \bar{d}^*)^2}} \tag{4.25}$$

where n is total number of sample points, $x_1, x_2, \ldots, x_n$; f_i represents the associated objective function value for the points with $\bar{f}$ being mean objective function value within the sample. The distance, d_i^*, between points is calculated as the distance between each point and the best point within the sample, with $\bar{d}^*$ representing the mean distance of the sample of points.

4.3.2.7 Gradient measures

The **gradient measures** [177] estimate the steepness of the gradients present within a problem search space. The gradients are determined between the sampled points within a progressive random walk, with the gradient determined between the points. A total of $n+1$ points are required to produce n gradients for the measures. Each gradient is calculated using

$$g(\mathbf{x}_i) = \frac{f(\mathbf{x}_{i+1}) - f(\mathbf{x}_i)}{\mathrm{d}(\mathbf{x}_{i+1}, \mathbf{x}_i)} \tag{4.26}$$

where $\mathrm{d}(\mathbf{x}_{i+1}, \mathbf{x}_i)$ is the Euclidean distances between the points.

Three different gradient metrics may be obtained using the calculated gradient information between the sampled points:

1. The **average gradient**, G_{avg}, provides an indication of the average gradient between neighbouring solutions within a random walk. Importantly, only the absolute gradient value is considered for this measure. The use of gradients is to quantify the perceived steepness within the optimisation problem search space. As a result the direction of the gradient does not matter and the absolute value is used in order to prevent the interference between positive and negative values with the mean calculation.

2. **Gradient standard deviation**, G_{std}, is an indication of how the measured gradients differ from the mean throughout the landscape walk. Lower deviation values would indicate that the average gradient is a good indicator of gradients within the problem search space. Large values are indicative of severe gradient changes which would be the result of *cliffs* and/or *valleys* being present within the search space. Another possibility would be large plateaus within the problem search space that differ from the rest of the search space.

3. The **maximum gradient**, G_{max}, provides the largest estimated gradient within a random walk. If G_{max} is larger than G_{avg} for a given random walk through the problem search space, then there are parts of the problem space that stand out from the remainder of the problem space. Larger values for both the average gradient and the maximum gradient indicate a highly rugged problem space.

4.4 Conclusion

This chapter discussed different DOP benchmark problems and introduced the notion that generator functions for dynamic environments are a feasible approach to create benchmark problems. The complexity of the dynamic optimisation problem landscapes was highlighted and that the problem instances pose a challenge when attempting to compare optimisation algorithm performances due to the non-deterministic change of problem landscapes. Of the available benchmark problems and generators, the MPB is the most popular generator function within literature with the most understood dynamism.

Adding problem constraints that also change over time to DOPs produces instances of DCOPs where the complexity of the problem is arguably the greatest for an optimisation algorithm. The current sets of benchmark problems provide the groundwork for developing more capable optimisation algorithms, but the benchmark problems are either developed for a specific purpose and are not general enough. On the other hand, the available benchmarks that do have an increased generality do not allow for a large number of decision variables, limiting the usefulness of these benchmark problems.

Optimisation problem landscapes produced from generator functions can differ based on control parameters of the benchmark instance generator and

due to the use of randomness within the generator function itself. Regardless of the observed differences between two problem instances generated in this way, the instances will display similar characteristics because the same generator function with the same control parameters was used. The MPB generator function is particularly interesting because it has previously been shown to allow for the derivation of a comprehensive set of of DOP instances [77]. The other generator functions are not able to represent such a comprehensive classification of optimisation problem instances.

In order to determine the complexities of an optimisation problem instance, the use of FLA was considered. Although the complexity of the DCOP problem instances is not fully understood, the landscape measures still remain the only way feasible current method to quantify the perceived complexity of an optimisation problem search space.

The next chapter will expand on the information obtained from the current benchmark problems, for both DOPs and DCOPs, in order to produce a generator function for DCOPs. The goal of this new generator function will be to address the shortcomings of the current benchmark problem instances and generator functions, whilst allowing for the representation of all possible problem instance combinations.

Chapter 5

Constrained Moving Peaks Benchmark

Testing leads to failure, and failure leads to understanding.

Burt Rutan

The number of DCOP problem instances that are possible is effectively infinite, with each problem instance presenting a different level of optimisation problem complexity. In order to attempt to understand the differences in optimisation problem complexity, classification schemes about the characteristics of problem instances have started to be defined. From these classifications, multiple DCOPs can be addressed based on the change patterns observed within the optimisation problem over time.

The previous chapter discussed the differences between unconstrained DOPs and DCOPs instances. From the current benchmarks and function generators, the larger quantity of benchmark problems are available for DOPs. For DCOP problem instances the available choices are far more restricted. Moreover, the complexity of combining optimisation problem constraints together with changing problem landscapes makes the generation, or definition, of these kinds of optimisation problems far more difficult.

This chapter attempts to address the difficulty of producing DCOP benchmark instances by building upon solutions found within benchmark function generators for DOPs. Section 5.1 proposes a new benchmark function generator to generate DCOPs. The analysis of an empirical study of the new proposed function generator for DCOPs is discussed in section 5.2. The chapter concludes in section 5.3.

5.1 Proposed Constrained Moving Peaks Benchmark

The MPB is an optimisation problem generator that has established itself within dynamic optimisation problem research [17, 18, 24–26, 29, 130, 156, 159, 201, 209, 248]. The problem classification of Duhain and Engelbrecht [77] (discussed in section 2.2.5) classifies dynamic optimisation problems into 27 different problem instance categories and uses the MPB to generate these problem instances. It should also be noted that the MPB is also capable of generating static problem environments. By not allowing an environment change to occur during the optimisation process, the problem environment remains constant. For this reason, the number of problem environments described within [77] can be extended to a total of 28 problem environments classifications.

When considering optimisation problem constraints, it is desirable to also have the constraint landscape of the optimisation problem be constructed in a manner that is similar to the objective landscape. Such a formulation of the constraint landscape allows for the possibility of varying the constraint landscape over time. By composing the generated objective and constraint landscapes together, the result is a landscape where the objective landscape and constraint landscape may change over time in complete isolation. The constraint and objective function landscapes are generated using the MPB problem generator. Landscape composition can therefore be performed by taking the difference between the objective and constraint landscapes as the MPB produces maximisation problem landscapes with several peaks. Obtaining the difference between the generated landscapes would produce a problem search space that includes infeasible regions. The resulting optimisation problem is defined as:

$$h(\mathbf{x}) = f(\mathbf{x}) - g(\mathbf{x}) \tag{5.1}$$

where f defines the MPB generated objective function landscape and g defines the MPB generated constraint landscape, and $\mathbf{x}$ is a location vector within the optimisation problem search space of the objective landscape.

Each of the generated landscapes may have their own configuration parameters (refer to section 4.1.3) which allow for a different number of peaks within the search space together with different peak modification and movement characteristics. In the event that the f and g MPB generators have different problem domains, it is recommended that the entities of the optimisation algorithm be initialised based on the domain of the objective function landscape, *i.e.*, the domain of the f generator function. The resulting composed optimisation problem will have a peak height range of $[-minHeight_g, maxHeight_f]$ and infeasible areas are indicated where $h(\mathbf{x}) < 0$. Due to the independence of the f and g generator functions, the frequency and the severity of change for each independent generator function need not be the same. Furthermore, it is

also recommended to have different streams of randomness for each generator function.

The composed generator function, h, has the following advantages:

- The problem and constraint problem spaces are independent from each other. The problem space may remain constant, whilst varying the constraint space, or any combination thereof.
- The dimensionality of the objective problem space and the constraint problem space need not match. For example, a more complex constraint problem space can be combined together with a less complex objective problem space.
- Plotting the generated composed problem space is trivial when using 2D and 3D visualisations.
- The composition allows for 28 environment types for both the objective problem space and the constrained problem space. As a result, there are a total of $28^2 = 784$ possible problem instances. The 28 different problem instances include the 27 problem instances as defined by Duhain and Engelbrecht [77] together with a static problem instance, labelled `STA`. To identify problem instances, the generator function h can label its problem instances using the form *objective space behaviour/constraint space behaviour* which represents the composition of the objective and constraint spaces. An example of this naming scheme is the `A1C/STA` problem instance where the objective space and constraint spaces respectively are the `A1C` and the `STA` problem instances.
- Inclusion of additional equality and inequality constraints to further constrain the resultant composed problem space is still possible.

The problem formulation of the CMPB problem generator addresses the criteria defined by Nguyen and Yao [208] which requires that:

1. The structure/percentage/shape of the feasible and infeasible regions within the resulting problem space changes over time.
2. Optima within the problem search space may appear in different disconnected regions.
3. Changing constraints may reveal better optima in a static problem space.

One aspect of the criteria of Nguyen and Yao [208] that is not directly addressed by the CMPB is the notion of time-linkage. Time-linkage requires that the evaluation of the algorithm on the optimisation problem may result in an environment change. Although CMPB does not directly cater for this requirement, the requirement may be considered by observing when the optimisation

algorithm evaluates the current problem landscape and then to respond to the evaluation.

Figure 5.1 illustrates a CMPB environment over nine environment changes, with the initial environment labelled as environment 0. The three dimensional environment is projected from a top-down view to show the interaction between the peaks of the constraint and objective MPB landscapes.

5.2 Landscape Analysis of the Constrained Moving Peaks Benchmark

This section examines the landscape characteristics of the CMPB problem generator using the landscape metrics introduced in the previous section. Section 5.2.1 explains the experimental setup for the investigation of the problem space environments that are generated by the CMPB generator function. The results of the experiments are presented in section 5.2.2 with the results indicating a favourable behaviour for the CMPB generator function.

5.2.1 Experimental Approach

In order to evaluate the problem characteristics of the CMPB generator function, only progressive random walks through the problem search space are considered to allow for the application of gradient-based metrics. A total of *numSteps* consecutive candidate solutions are sampled during the progressive random walk which begins from a random starting point on the boundary of the problem search space.

From an initial, randomly generated problem environment a total of 30 independent random walks are sampled. Thereafter, the problem environment will experience a change and a new problem search space will be generated, derived from the current problem landscape generation parameters. The candidate solutions points within the independent random walks are evaluated once again on the updated problem landscape. This process is repeated for 10 consecutive landscape changes, for each of the 30 independent random walks.

To ensure that there is complete fairness between problem landscape comparisons, the random walk sampling process as well as the problem environment changes are managed by the CIlib software library. CIlib (discussed in part IV of this thesis) enables the exact reproduction of environments and walks across all the environment changes. Without the ability to exactly reproduce the problem environments and random walks, the comparisons would be unfair (possibly even invalid) as the problem environments and the independent random walks cannot be guaranteed to agree.

An initial random seed value of `123456789L` was used from which the random number generators for the objective and constraint spaces were initialised. By duplicating the walks for each problem environment a representative sample

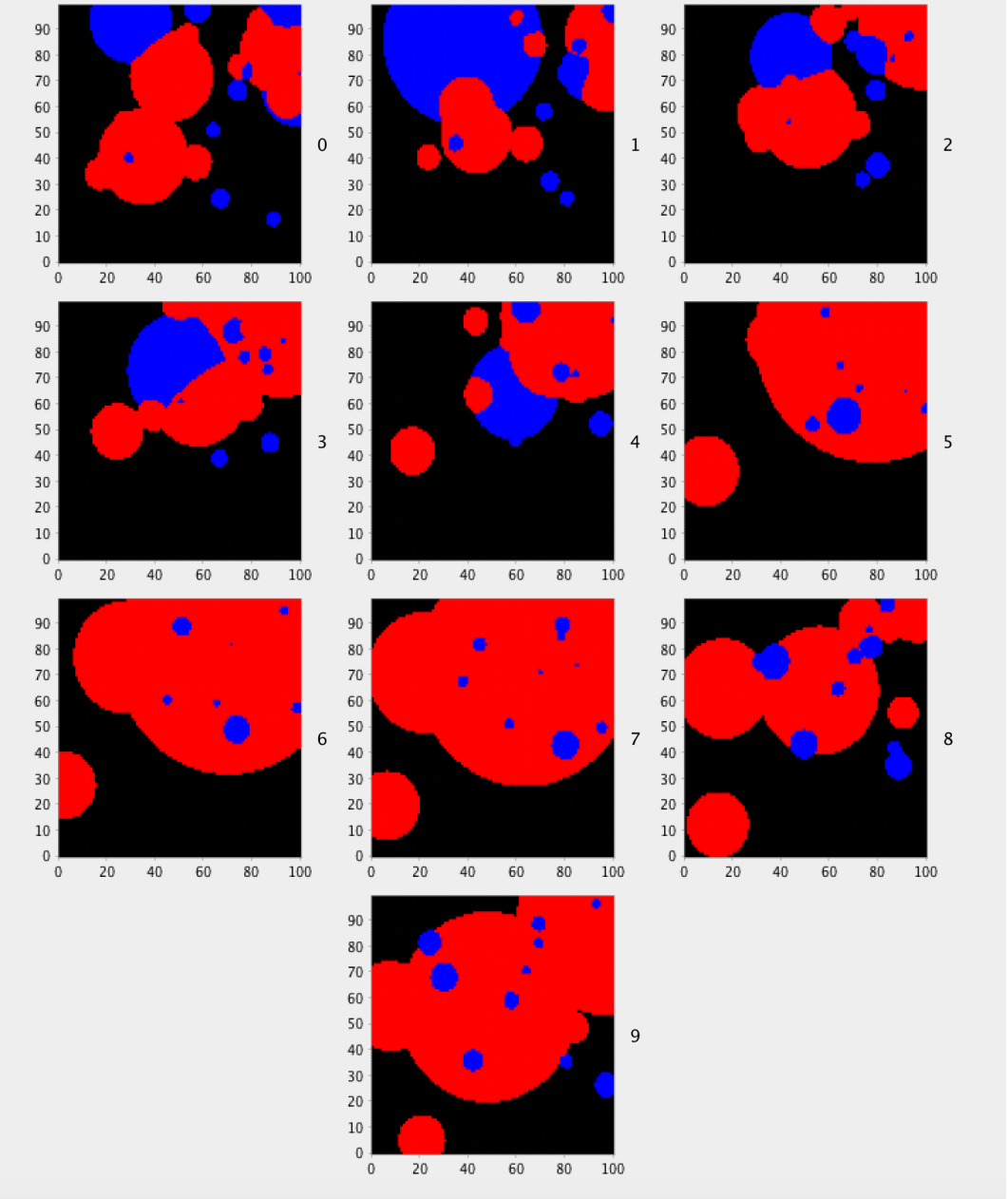


Figure 5.1: Constrained MPB instance with nine consecutive environment changes. Red coloured regions are infeasible ($h(x) < 0$). Black regions indicate feasible regions without solutions ($h(x) = 0$); Blue regions have solutions present ($h(x) > 0$).

is possible by limiting the differences to the underlying problem environment alone. Figure 5.2 illustrates the mean fitness landscape metrics for each of the 10 problem instances with the top-down projection onto the xy plane for the same problem instances, given in figure 5.1.

5.2.2 Landscape Analysis Results

The FSR metric for the environments in figure 5.2 displays the ratio of feasible space across the problem instances. The percentage of feasible space ranges from 0.38 to 0.84, which is seen as the grey and white search space regions in the projection images within figure 5.1. As a result, the changes to the problem environment allow the problem search space to transform from predominantly feasible to predominantly infeasible when considering the composed landscape produced by the CMPB problem generator. Furthermore, the regions of infeasible space change over time in both size and connectedness as the environment changes.

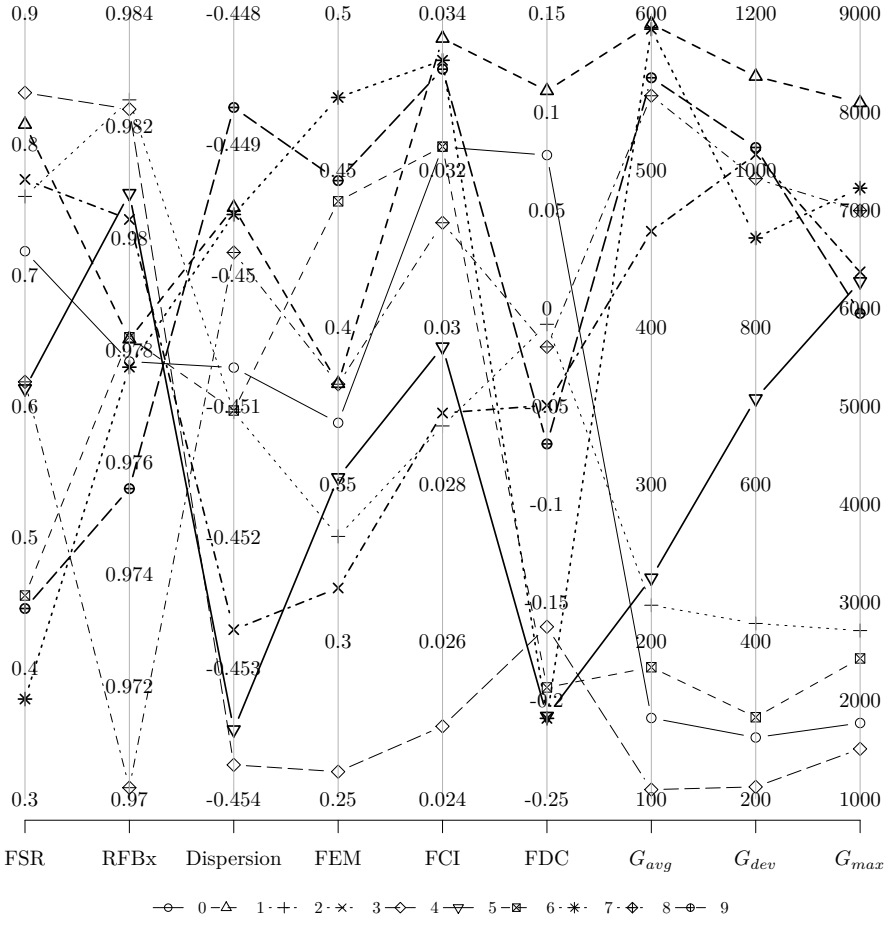
The RFBx metric displays minimal variation in the obtained values, as illustrated in figure 5.2. This minimal variation indicates that feasible regions are not disjoint. The problem landscapes of figure 5.1 confirm that the metric does not indicate that the generated problem spaces actually do contain disjoint feasible regions. The domain of the generated problems is large and the random walks did not traverse the problem search space in such a way that these regions were identified. A combination of a larger number of random walks and/or longer sampling lengths would result in a larger coverage of the problem search space and should eventually encounter these disjoint feasible regions.

The dispersion metric produces a similar value for all of the generated problem landscapes. The dispersion value is indicative of a good spread of solutions within the problem space which provides a fair probability for an optimisation algorithm to locate solutions.

The ruggedness of the generated problem spaces, as provided by the FEM metric, indicates that the generated problem spaces vary from marginally to slightly rugged. These values confirm the expectations for the generated MPB problem landscapes because the MPB has a base being a flat plane with peaks that are super-imposed into the problem space. With a larger number of peaks as input to the problem generator, an even more rugged landscape will be generated.

The searchability of the generated problem spaces, as provided by the FCI metric, indicates that the generated problem landscapes do not aid in the searchability of the problem itself. This is an expected result as the disjointedness of the feasible regions, both in and outside of infeasible spaces, do not guide an optimisation algorithm to a better solution.

The FDC metric shows that the generated problem landscapes display correlations close to 0. A correlation of 0 indicates that marginal correlations exist between the observed fitness values and the distance to optima. The FDC



RFBx = Ratio feasibility boundary crossings
FEM = First entropic measure
FCI = Fitness cloud index
FDC = Fitness distance correlation
G_{avg} = Average walk gradient
G_{dev} = Walk deviation
G_{max} = Maximum gradient in walk

Figure 5.2: Fitness landscape characteristics of a dynamic, constrained optimisation problem

values suggest that the information present in the problem search space does not aid an algorithm to a single optimum value but instead presents unbiased alternate solutions.

The gradient metrics ($G_{avg}, G_{dev}, G_{max}$) all display a similar trend of generally larger gradients. The results are not surprising and are expected based on the definition of the MPB generator function. The MPB generator makes the generation of large peaks with narrow bases probable as the peak generation requires the sampling of random numbers during the peak generation process. The gradient metrics all display large values which is indicative of large gradient changes in the problem space.

From the results the changes to the environment show no discernible pattern in the calculated landscape metrics. Furthermore, the absence of a pattern is also missing when visually inspecting the resulting environments within figure 5.1.

5.3 Conclusion

This chapter proposed a new benchmark function generator, the CMPB, to create DCOP benchmark instances. The new proposed generator function allows for the generation of both the objective search space and the constraint search space, before these landscapes are composed together to produce the final optimisation problem search space. The CMPB generator function addresses the required criteria of DCOP benchmark problems, whilst removing dimension restrictions visible in other DCOP instance generators. Moreover, the CMPB problem generator can provide a wide selection of problem instances by extending the comprehensive classification [77] for DOPs.

The quality and characteristics of the resulting problem instances from the CMPB function generator were tested using FLA and different measures. From the experimental results, it was shown that the problem instances produced form the CMPB function generator are diverse and complex. Furthermore, it was shown that the resulting problem instances do not expose attributes and characteristics of the optimisation problem which may be exploited by an optimisation algorithm. The result is a DCOP function generator which can not only provide diverse problem instances, but the explicit generation of benchmark problems for each of the SOSC, SODC, DOSC and DODC categories of DCOP.

Chapter 6

Performance Measures

Science cannot progress without reliable and accurate measurement of what it is you are trying to study. The key is measurement, simple as that.

Robert D. Hare

Previous chapters introduced the relationship between the optimisation problem, optimisation algorithm, and problem space constraints. Dynamic optimisation algorithms attempt to locate and to track solutions to optimisation problems whilst considering the problem space constraints. Each of the above mentioned aspects of the optimisation process may independently be either static or dynamic in nature.

Section 6.1 discusses the need and requirements for performance measurement within DOPs. Performance measures for dynamic optimisation algorithms are discussed in section 6.2 by sub-dividing available measurements into different categories before discussing the shortcomings of performance measures. A new vector-based performance measure, which defines the algorithm performance profile, is proposed in section 6.3. Final remarks for the performance measures are presented in section 6.4.

6.1 Performance Measurement

The no free lunch theorem (NFL) [304] states that no single optimisation algorithm is applicable to all optimisation problems and more appropriate algorithms should be used for specific problems when possible. Measuring the effectiveness or performance of an algorithm is an established and well-understood practice for SOPs [85]. However, measures applicable to SOPs are not always appropriate for use within DOPs [51, 130, 186]. Performance measures generally produce a single value for consideration when comparing the performance of different algorithms. As a result, no single performance measure

can quantify a dynamic optimisation algorithm unambiguously, often necessitating multiple measures to explain optimisation algorithm behaviour [299].

The importance of accurate and precise measurement has already been established in other scientific fields, such as chemistry. Therefore, performance measurement within CI should be regarded with the same importance in order to determine the most accurate results. This becomes more evident as the use of CI increases.

6.2 Performance Measures for Dynamic Optimisation Algorithms

Performance measures for DOPs may be grouped into two categories: optimality or behaviour based measures. Optimality-based measures are discussed in section 6.2.1 and focus on the quality of solutions located by the optimisation algorithm. Section 6.2.2 discusses behaviour-based measures which focus on how changes to the problem landscape impact the behaviour of the optimisation algorithm. The shortcomings of the performance measures found within existing literature are identified in section 6.2.3.

6.2.1 Optimality-Based Measures

Optimality-based performance measures focus on how accurate the solution found by an optimisation algorithm is. Additionally, optimality measures allow for a simple comparison of algorithm performance on DOPs [209].

When the optima of a problem search space are known ahead of time, found solutions may be compared to the known best solution. The difference between the known solution and the optimisation algorithm solution produces an error value for the observed solution. The error value can only be determined if the optimal solution is known across all problem change periods. If it is not possible to determine the error in the quality of the produced solution, the performance measures may instead measure the best current solution from the current set of candidate solutions. The sub-sections that follow discuss the most popular optimality-based performance measures.

6.2.1.1 Best of Generation

The best of generation (BOG) fitness records a series of fitness values for the best candidate solution. The sampled fitness value is defined as

$$P_{\text{BOG}}(t) = f(\mathbf{x}^*(t), t) \tag{6.1}$$

where $\mathbf{x}^*(t)$ is the best candidate solution found by the optimisation algorithm at iteration $t \in \{1, 2, \ldots, n_t\}$, and n_t is the total number of algorithm iterations. P_{BOG} may be plotted against the algorithm iteration count to create

a performance plot for the algorithm execution. The average BOG may also be considered as an indication of the average best fitness value for the best candidate solution over the course of the algorithm execution. The average BOG measure is

$$\overline{P}_{\text{BOG}} = \frac{1}{n_t} \sum_{t=1}^{n_t} P_{\text{BOG}}(t) \tag{6.2}$$

P_{BOG} is closely related to the collective mean fitness (CMF) proposed by Morrison [200], and by extension the collective mean error (CME).

6.2.1.2 Collective Mean Fitness

The CMF [200] is the mean fitness value of the best candidate solution, over all algorithm iterations. The measure is defined as

$$P_{\text{CMF}} = \frac{1}{n_t} \sum_{t=1}^{n_t} f(\mathbf{x}^*(t), t) \tag{6.3}$$

where $\mathbf{x}^*(t)$ is the best candidate solution at iteration t and n_t is the total number of algorithm iterations.

6.2.1.3 Collective Mean Error

The CME [200] records the mean error of the best candidate solution over the entire optimisation algorithm execution. Error values are the difference between the current best candidate solution fitness value and the objective function optimum value. The measure is defined as

$$P_{\text{CME}} = \frac{1}{n_t} \sum_{t=1}^{n_t} err(\mathbf{x}^*(t), t) \tag{6.4}$$

where n_t is the number of iterations within an optimisation algorithm execution, and err is a function that calculates the error value at algorithm iteration t for the current best candidate solution, $\mathbf{x}^*(t)$.

The CME is identical to the CMF measure, except for the use of the error value instead of the raw objective function value. The CME is regarded as a good overall measure [200] to quantify optimisation algorithm performance on a dynamic optimisation problem.

6.2.1.4 On-line Performance

The on-line performance (OP) [23, 63] considers the average fitness of all candidate solutions for all fitness evaluations, over the entire algorithm execution. The measure is defined as:

$$P_{\text{OP}} = \frac{1}{n_{fe}} \sum_{e=1}^{n_{fe}} f(\mathbf{x}^e, e) \tag{6.5}$$

where $\mathbf{x}^e$ is the candidate solution considered in fitness evaluation e of the optimisation algorithm.

The OP result will have a larger value when all candidate solutions are similar; maximising the value returned by equation (6.5). DOPs require a diverse set of candidate solutions in order to locate and track solutions. Within such a diverse set of candidate solutions the likelihood of candidate solutions with poor objective function values increases. As a result, the online performance measure provides little to no benefit in understanding algorithm performance within DOPs.

6.2.1.5 Modified Off-line Error

The modified off-line error (MOE) [23, 27] observes the average error of the best performing candidate solution since the last change of the problem landscape. The measure is a cumulative value which resets upon the problem experiencing change and is calculated as

$$P_{\text{MOE}} = \frac{1}{n_{fe}} \sum_{e=1}^{n_{fe}} err(\mathbf{x}^e_{best}, e) \tag{6.6}$$

where n_{fe} is the total number of objective function evaluations and $err(\mathbf{x}^e_{best}, i)$ is the error of the current best candidate solution, $\mathbf{x}^e_{best}$, since the last landscape change.

Knowledge of the optimisation problem search space is required, including knowledge of later problem landscapes. The knowledge of the landscape is needed in order to provide a value for the global optimum value from which the error between the current best candidate solution and search space optimum is calculated. As with other measures, the value produced by the offline error is not normalised, allowing bias within the optimisation problem search space to change periods with large error values.

6.2.1.6 Modified Off-line Performance

The modified off-line performance (MOP) [23, 27] is identical to the MOE measurement, but does not require exact knowledge of the global optimum within the problem search space. This is also true for later problem landscapes that are the result of the problem changes, originating from the original problem search space. The MOP prefers the quality of the current best candidate solution, within the current iteration, as the measurement result. The measure is calculated as

$$P_{\text{MOP}} = \frac{1}{n_{fe}} \sum_{e=1}^{n_{fe}} f(\mathbf{x}^e_{best}, e) \tag{6.7}$$

6.2.1.7 Relative Error

The *relative error* [91, 299] (or *optimisation accuracy*) normalises the fitness of the best candidate solution over the minimum and maximum values for the problem. The result is a single value between $[0, 1]$ representing the algorithm performance for the current iteration t. Values close to 0 indicate poor algorithm performance, whereas values close to 1 indicate good algorithm performance. The measure is formally defined as

$$P_{\text{RE}}(t) = \frac{f_{\text{best}}(t) - f_{\text{min}}(t)}{f_{\text{max}}(t) - f_{\text{min}}(t)} \tag{6.8}$$

where $f_{\text{best}}(t)$ is the best candidate solution in iteration t, and $f_{\text{min}}(t)$ and $f_{\text{max}}(t)$ are the minimum and maximum quality values a candidate solution may have.

The relative error is similar to the BOG measure but has the advantage of normalising the resultant value. Furthermore, the relative error biases less towards large errors in problem landscape change periods. Problem search space knowledge (across change periods) is a prerequisite for this measure.

6.2.1.8 Normalised Score

The *normalised score* [206, 207] allows for the comparison of multiple algorithms across multiple problem instances, even within a dynamic problem search space. The resulting performance for a test instance j would be normalised to the range $[0, 1]$. The best performance is assigned the value of 1 and the worst is assigned the value 0. The overall performance of an algorithm is the average of the normalised scores. By averaging the normalised scores, a better performing algorithm (which should have received more "wins" when compared to the other considered algorithms) will have an average score value closer to 1. Similarly, the worse performing algorithm will have an averaged score value closer to 0. The normalised score for algorithm i is calculated as

$$P_{\text{NS}} = \frac{1}{n_c} \sum_{c=1}^{n_c} \frac{|e_{\text{max}}(c) - e(i,c)|}{|e_{\text{max}}(c) - e_{\text{min}}(c)|}, \forall i = 1, \ldots, n_h \tag{6.9}$$

where $e(i, c)$ is the MOE for algorithm i in the change period c of the problem instance; $e_{\text{max}}(j)$ and $e_{\text{min}}(j)$ respectively are the largest and smallest MOE across all algorithms for each problem landscape c, and n_h is the number of algorithms being compared. The normalised score is similar to the P_{RE} measure, with the differences being the averaging process on the normalised values and the choice of value to use within the normalisation process. The normalised value may be any measured value for the algorithm performance including candidate solution fitness; *i.e.*, the BOG value. When the measure was proposed [206, 207], there was no mention how randomness would influence changes to the problem landscape, nor how randomness sampling would

impact the results obtained from equation (6.9). The comparison ability of P_{NS} across algorithms and problems is therefore only possible if the initial problem landscapes together with all landscape changes are identical and do not interfere with the optimisation algorithm. Conversely, the optimisation algorithm execution should also not interfere with the optimisation problem landscape changes. An example of benchmark problems that observe a deterministic landscape change is the G24 set of benchmark problems [206]. The changes within these benchmark problems are a function of the number of algorithm iterations (see section 4.2.2).

6.2.1.9 Window Accuracy

Weicker [299] quantified the accuracy of a dynamic optimisation algorithm as a ratio. The ratio compares the current iteration performance to a window of the previous ω algorithm iterations. The difference between the current best and worst candidate solutions is compared to the difference of the best and worst candidate solution fitness values found within the specified window. The measurement is defined as (assuming maximisation):

$$P_{\text{WA}}(t) = \frac{f(\mathbf{x}^*, t) - w_{\text{worst}}(t)}{w_{\text{best}}(t) - w_{\text{worst}}(t)} \tag{6.10}$$

with

$$w_{\text{best}}(t) = \max\{f(\mathbf{x}_{\text{best}}(t'), t') \mid t - \omega \leq t' \leq t\} \tag{6.11}$$

$$w_{\text{worst}}(t) = \min\{f(\mathbf{x}_{\text{worst}}(t'), t') \mid t - \omega \leq t' \leq t\} \tag{6.12}$$

where $\mathbf{x}^*$ is the best candidate solution at time step t; $\mathbf{x}_{\text{best}}(t)$ and $\mathbf{x}_{\text{worst}}(t)$ are functions to respectively determine the best and the worst candidate solution fitness values at the time step contained within the window.

6.2.2 Behaviour-Based Measures

Behaviour-based measures examine an optimisation algorithm's response to change in the problem landscape. The information provided by behaviour-based measures is useful to explain algorithm behaviour within a problem change period, or the behaviour as a result of a landscape change. As a result, behaviour-based measures together with optimality-based measures will provide a more complete indication of the algorithm behaviour and performance. The sub-sections that follow discuss commonly used measures which focus on the recovery behaviour of dynamic optimisation algorithms within DOPs.

6.2.2.1 Balancing Exploration and Exploitation

As a DOP undergoes change, a dynamic optimisation algorithm should locate new and track existing solutions. In order to locate new solutions when the

problem landscape changes, the algorithm should be able to explore the changed problem search space. As a result, it is desirable to maintain a level of diversity within the set of candidate solutions to achieve the required exploration of the problem search space. The amount of diversity should also not prevent the refinement of problem solutions, requiring a balance between exploration and exploitation. One of the more common diversity measures is the mean diversity measure [218], even though other diversity measures also exist [34, 85, 218]. For example, the centroid of the diversity calculation may be changed to use the median instead of the mean for the j-th dimension value.

The *percentage feasibility* is a measure of the number of solutions within feasible and infeasible regions of the problem search space for each algorithm iteration. Plotting the percentage of feasible solutions over the entire algorithm execution indicates whether an algorithm is able to direct the search process away from infeasible spaces or not. Expanding the idea to be the average of a number of algorithm executions (whilst ensuring that the problem landscapes are exactly the same) may indicate how reliable or robust an algorithm is for a specific instance of an optimisation problem.

The *recovery rate* for an optimisation algorithm determines how quickly the algorithm is able to start converging onto optima (global or local) before the next problem landscape change. Faster algorithm recovery indicates that the algorithm is both robust and stable. The criteria for recovery is not limited only to solution quality. In some scenarios it may be more beneficial to achieve solutions that are at least feasible, regardless of the solution quality. Algorithm recovery determines if the algorithm is able to effectively explore and/or exploit regions of the problem search space after a problem change, indicating that the algorithm is able to adapt to landscape changes.

6.2.2.2 Average Best Error Before Change

The average best error before change (ABEBC) measure highlights the *exploitative capability* [279] of an optimisation algorithm. In other words, whether the solution found by the optimisation algorithm just before the problem landscape changes a "good" solution. Knowledge of the global optimum within the problem search space and for all problem landscape change periods, is a prerequisite to use this measure. The measure is formally defined as

$$P_{\text{ABEBC}} = \frac{1}{n_c} \sum_{c=0}^{n_c} err(\mathbf{x}^*, t_c) \tag{6.13}$$

where the error in fitness value of the current best candidate solution $\mathbf{x}^*$ and the global optimum is calculated at algorithm iteration t_c, which is the iteration where the problem landscape change c occurs; n_c is the total number of landscape changes experienced by the optimisation problem.

The average best error before change (ABEBC) has the same concern as the BOG measure, where the values obtained are not normalised, potentially biasing the measure towards change periods with large errors.

6.2.2.3 Average Best Error After Change

Optimisation algorithm *stability* [279] is the ability for an optimisation algorithm to maintain a solution after the problem landscape changes. The average best error after change (ABEAC) measure is identical to the ABEBC measure with the exception that the measurement is taken directly after a problem landscape changes, and not before the change occurs. The measure is defined as:

$$P_{\text{ABEAC}} = \frac{1}{n_c} \sum_{c=0}^{n_c} err(\mathbf{x}^*, t_c + 1) \tag{6.14}$$

where err is the function calculating the observed error value against the current best candidate solution $\mathbf{x}^*$ and t_c is the algorithm iteration wherein the problem landscape changed.

The concerns of the ABEBC measure are also applicable to the average best error after change (ABEAC) measure. Such concerns include value normalisation, knowledge of the landscape global optima for all change periods and uncertainty regarding solution refinement.

6.2.2.4 Lowest/Highest Best Error Before Change

The lowest best error before change (LBEBC) and highest best error before change (HBEBC) [161] respectively record the lowest and highest error values for the best candidate solution across the problem landscape change periods. The measurements are counterparts to the ABEBC and are defined as

$$P_{\text{LBEBC}} = \min_{c=1,2,\ldots,n_c} err(\mathbf{x}^*, t_c) \tag{6.15}$$

$$P_{\text{HBEBC}} = \max_{c=1,2,\ldots,n_c} err(\mathbf{x}^*, t_c) \tag{6.16}$$

where n_c is the number of problem landscape changes experienced by the optimisation problem, $\mathbf{x}^*$ is the current best candidate solution and t_c is the iteration where the landscape change c occurs.

6.2.2.5 Absolute Recovery Rate

The absolute recovery rate (ARR) [208] is a measure to calculate the recovery rate of an optimisation algorithm after a problem landscape change. The ARR is calculated as

$$P_{\text{ARR}} = \frac{1}{n_c} \sum_{i=1}^{n_c} \frac{\sum_{j=1}^{p(i)} (f_{best}(i,j) - f_{best}(i,1))}{p(i)[f^*(i) - f_{best}(i,1)]} \tag{6.17}$$

where $f_{best}(i,j)$ is the fitness of the best candidate solution since the last problem landscape change, n_c is the number of landscape changes, $p(i)$ is the number of iterations within the problem landscape change period i, and $f^*(i)$ is the global optimum value for the change period. The ARR results in a value within the range $[0,1]$, where a value of 1 indicates that the algorithm was able to locate and converge onto a global optimum. As with the ABEBC and ABEAC measures, knowledge of the global optimum value, for each problem landscape change, is required in order to use the ARR measure.

The ARR provides an indication of the change in algorithm performance across the change period of the optimisation problem. This measure is not, however, an unambiguous indication of algorithm improvement and may present poor results when an algorithm begins and ends the change period near a good candidate solution. Therefore, the ARR results must be considered in conjunction with other results.

6.2.3 Shortcomings of Existing Performance Measures

Development of performance measures occurs in order to improve the investigation and explanation of specific algorithm behaviour questions. As a result, each measure may require additional information to produce a value. An optimisation algorithm may then be scrutinised based on the measurement results. Importantly, no performance measure is perfect for all scenarios and the shortcomings of measurements should be considered. The sub-sections that follow broadly categorise these shortcomings.

6.2.3.1 Problem Search Space Knowledge

Knowledge of the problem search space may be required for the performance measurement to calculate the performance value. This may include the maximum and minimum values for the optima, possibly together with optima positions within the problem search space. For error based measurements, which compare the fitness of a candidate solution to a target optima value, problem landscape information is a requirement. Within DOPs, optima information across all landscape changes may be required. The inclusion of problem constraints in DOPs further increases the need for domain knowledge in order to compensate for regions of the problem landscape becoming infeasible.

6.2.3.2 Outlier Sensitivity

Performance measures that aggregate values to produce a summary statistic are sensitive to outliers. Inclusion of outliers within summary statistics produces a distorted view of the sampled optimisation algorithm performance. If an optimisation algorithm uses aggregate values to guide the search process, the presence of outliers may impede the optimisation algorithm's search behaviour.

The calculation of an average (the mean) is an example of where outliers interfere with the aggregate result. Additionally, scenarios that consider the maximal and minimal values from a list are also affected by outliers.

6.2.3.3 Cardinality of Output

Performance measurements may provide a single scalar value to quantify the performance measurement of an optimisation algorithm. The alternative is that the performance measurement produces multiple data values. Scalar values are simpler to work with, particularly within statistical tests, because a single value is the representation of algorithm behaviour. Multiple values represent a vector of time-series data which is indexed either by the algorithm fitness evaluations or algorithm iterations.

Scalar values unfortunately also have the disadvantage that the value represents a series of values that have been aggregated. Common scalar values include aggregates such as the mean, standard deviation and variance. The most simplistic of these aggregations is the mean calculation. As previously mentioned, the mean is sensitive to the presence of outliers, but it is also insensitive to the specific scenario that the final result represents. Consider the following simplistic example where an algorithm performance measure produces a series of four unique values. In order to compare the algorithms, the produced series data must be reduced into a scalar value using the mean calculation. Two of the samples within the experiment data are the data arrays $A = [3, 3, 3, 3]$ and $B = [6, 0, 6, 0]$. Calculating the mean of these samples produces the same aggregate value of 3. By only considering the mean of the samples, the two algorithms are considered as having equivalent performance for the specific problem in the experiment. Data array B has a much larger variance than data array A and standard deviations are often used in conjunction with the mean in publications for exactly this reason. The standard deviation together with the mean provides a more complete picture of algorithm performance. As a result, multiple aggregated scalar values are used in order to interrogate algorithm performance.

It was noted by Cruz *et al.* [51], Moser and Chiong [201], and Nguyen *et al.* [209] that a single aggregate value, determined at the end of the algorithm execution, is insufficient to quantify the performance of an algorithm on a DOP instance. Instead, the general recommendation is to measure performance multiple times during algorithm execution.

6.2.3.4 Intuitive Understanding

Performance measures generally produce unit-less numbers which are not understood outside of the required context. The fitness of a candidate solution makes little sense unless it is placed within the context of the current optimisation problem. It is the optimisation problem that defines the optimisation

scheme (either minimisation or maximisation) for the optimisation algorithm. The optimisation scheme determines if the produced fitness values are "good" or "bad". The uncertainty of what a result suggests increases when the optimisation problem changes over time.

An example of a performance measure that does not need to know about the problem context is the error value, particularly the normalised error. Error values are defined such that the best achievable result is that there is no error observed. With no observed error, the obtained solution is the target solution for the optimisation problem. Conceptually, error values encode the optimisation target (minimisation or maximisation) during their calculation.

6.2.3.5 Susceptibility to Undefined Values

Ideally, the result of a performance measure should never be "undefined". Undefined values, such as those produced within division with a zero denominator or the square-root of a negative number, may not be noticed and could produce results that make no sense during comparisons. Different strategies are available to mitigate, and possibly remove the occurrence of such errors. It is, however, not a certainty that measures are capable of accepting these invalid inputs. An example of a failure would be a performance measure returning the *special* value of `NaN` (not a number). The IEEE specification for floating-point numbers makes provision for a special "not a number" value which indicates an error, yet is itself still regarded as a valid number. As a result, number calculations that use a `NaN` as input, produce `NaN` as output and complicates the process of locating the point of failure. Not all computing platforms allow for the representation and use of `NaN`, but a large enough number do allow its use to justify the importance of ensuring correct and robust handling of undefined values.

Performance measures should be implemented such that invalid results are not possible. However, when the possibility of an undefined result exists, it should be recorded such that subsequent calculations are able to operate with these missing values. Additionally, the cause of such errors may be the result of an implementation error within the algorithm but manifest within the performance measure.

6.2.3.6 Interpretation of Maximisation and Minimisation Problems

Performance measures may require different equations for minimisation and maximisation problems. As a result, the possibility exists for the incorrect formulation of the measurement calculation to be used, yielding incorrect results. This is particularly evident when the optimisation scheme is required after the algorithm execution within additional analytical processes. On the other hand, performance measures producing a normalised error value do not require knowledge of the optimisation scheme because the error value implicitly

incorporates the notion of "good" and "bad". Error values are still, however, sensitive to and reliant on the optimisation scheme to calculate the error value. Ultimately, additional care is needed when using performance measures that are sensitive to the optimisation scheme. An example of how measurements can be misinterpreted based on the optimisation scheme, is the *sampled relative error* of Li *et al.* [161]. In the case of a minimisation problem, the sampled relative error is calculated as

$$r(t) = \frac{F(\mathbf{x}^*(t))}{F(\mathbf{x}_{best}(t))} \tag{6.18}$$

with the sampled relative error for a maximisation problem calculated as

$$r(t) = \frac{F(\mathbf{x}_{best}(t))}{F(\mathbf{x}^*(t))} \tag{6.19}$$

where $F(\mathbf{x}_{best}(t))$ is the fitness value of the best found solution and $F(\mathbf{x}^*(t))$ is the fitness value of the optimum at time-step t. The interpretation of $r(t)$ increases hyperbolically for minimisation problems, but increases linearly for maximisation problems as $F(\mathbf{x}_{best}(t))$ approaches $F(\mathbf{x}^*(t))$ [73]. As a result, the meaning derived from the same measurement differs based on the optimisation scheme. Additionally, the sampled relative error also allows for the production of undefined values if either $F(\mathbf{x}_{best}(t))$ or $F(\mathbf{x}^*(t))$ are zero.

6.2.3.7 Landscape Scale Changes

When dramatic landscape changes occur within an optimisation problem, the value of optima may be altered in a severe fashion. The calculated error value to target optima will represent a different percentage change based on the scale of the underlying problem search space. For example, when considering an optimisation problem landscape before and after a problem change where an optimum transforms from the lowest to the largest possible value. In this scenario, the possible range for the optima value is $[30, 70]$. Before the landscape change the impact of the observed raw error, e, is denoted by the ratio $\frac{e}{30}$, whereas after the change the same error ratio would be $\frac{e}{70}$. As a result, the change of problem landscape has made the error before the landscape change less severe within the updated problem landscape. Therefore, the use of raw error or fitness values can produce confounding statistical results as the problem landscape changes.

6.2.3.8 Variance Over Time

Optimisation algorithms may present as equivalently performing algorithms because the algorithms achieved similar final results. Unfortunately, some performance measures do not consider if an algorithm was experiencing erratic behaviour before achieving the final result. In fact, the algorithm may actually

still be experiencing volatile behaviour and similar performance results are simply a coincidence based on algorithm execution termination criteria. For example, all algorithm executions are terminated after the same number of iterations.

As a result, multiple performance measures must be considered simultaneously in order to quantify the algorithm behaviour. An example is the previously mentioned case where the mean algorithm fitness together with the standard deviation are considered to describe algorithmic behaviour and performance.

6.2.3.9 Parameters

When parameters are introduced into performance measures, the overall optimisation complexity increases. Performance measure parameters control the behaviour of the performance measure used to evaluate an optimisation algorithm. Multiple questions are necessary in order to determine the validity of the performance measure for the current optimisation algorithm and problem. For example, when performance measure parameters are required, it is uncertain if the selected parameter values are optimal for the current optimisation algorithm. Furthermore, would the optimality of measurement parameters require optimisation themselves, or is the selected value reasonable? As a result, minimisation of the number of configurable parameters is always preferred to optimising parameters.

6.2.3.10 Sensitivity to Optimisation Algorithm Execution

Population-based optimisation algorithms may execute in two distinct ways when considering the algorithm iterations. The synchronous iteration processes the entire collection of candidate solutions simultaneously, producing a new candidate solution collection as a result. Asynchronous iterations update each candidate solution independently, realising an updated objective function value for the candidate solution immediately. Performance measures that consider the entire collection of candidate solutions should consider the effect of the different algorithm iteration strategies, or when every objective function evaluation is considered.

6.2.3.11 Assumed Normality of Data

Studies have shown that parametric statistical tests are frequently the wrong choice for CI algorithms [68, 96, 97]. Parametric statistics have a clear assumption that the underlying data is normally distributed. For example, the cumulative mean of calculation for an algorithm expects a normal distribution. If the data is not distributed normally, a skewed perspective will be presented for the algorithm performance.

6.3 Vector-Based Measures

A number of measures mentioned in sections 6.2.1 and 6.2.2 (such as the P_{BOG} measure) do not result in a single scalar value, but instead yield a vector of values as the result of the measurement process. The measurement frequency determines the number of dimensions within the resulting vector. The most granular level of measurement produces a value for each iteration of the algorithm. This section describes how performance metric vectors allow for a more representative comparison of optimisation algorithms, provided that all other problem parameters and settings are identical. Section 6.3.1 discusses the problems with scalar-based performance measures, whilst section 6.3.2 introduces the notion of an algorithm performance profile. Section 6.3.3 highlights how the measure is also applicable for static optimisation problems, whilst section 6.3.4 relates the applicability of algorithm performance profiles within DCOP search space.

6.3.1 Problems with Scalar Value Comparisons

Shortcomings of performance metrics used to compare optimisation algorithms have been discussed in section 6.2.3. Scalar values are however still attractive because the single value comparison is simple to use. Unfortunately, the largest disadvantage with the use of scalar performance metrics is that the aggregation functions which produce the final metric result often apply a "smoothing" effect to the calculation. The simplest example of smoothing is the calculation of the mean, where the vectors A and B within section 6.2.3.3 produce the same result although the vectors are different. As a result of this smoothing, multiple scalar results must be used together in order to explain the performance behaviour of an algorithm.

In an attempt to overcome the problem with smoothing, Helbig and Engelbrecht [116, 117] proposed a measurement scheme which considers individual change periods for a DOP instance. The measurement scheme compares the performance measures between algorithms and across change periods by performing a Kruskal-Wallis test [150] to determine if a statistically significant performance difference exists. If the Kruskal-Wallis test indicates a significant difference, a pair-wise Mann-Whitney-U rank sum test is performed together with Holm correction [121] to limit statistical type-I errors (*i.e.*, false positives). Based on the outcome of the test, an algorithm is assigned a "win" or a "loss". Wins are given the numerical value of 1 whilst losses are assigned the value of -1; ties are rewarded with a value of 0. A *wins-minus-losses* score is assigned to each algorithm for the DOP instance. For a given measurement taken within a problem change period, the wins-minus-losses score is normalised as:

$$norm(p, m) = \frac{\sum_{i=1}^{n_c} (w_{i,m} + l_{i,m})}{n_c} \tag{6.20}$$

where $norm(p, m)$ is the average wins-minus-losses value for the problem p, for measurement m. The wins-minus-losses process does not, however, address the smoothing effect introduced by averaging the wins and losses achieved by an algorithm. As a result the final wins-minus-losses value is the central tendency value as the overall performance of the algorithm across change periods and problem instances. The final result of the measurement scheme remains a singular scalar value, with which the performances of algorithms is compared.

The wins-minus-losses approach is also sensitive to the shape of the underlying problem instance. Specifically, for the comparisons to be fair, the changes experienced within the problem must produce identical problem search spaces to allow for fair algorithm performance comparisons. Importantly, the comparison scheme cannot enforce that all problem landscapes within a DOP are identical and should instead be regarded as an error in the experiment formulation. Violation of the change period equivalence results in irreconcilable algorithm comparisons, because more variances are present than what is accounted for.

6.3.2 Proposed Measure for Algorithm Performance Profiles

In this section, vectors of performance measures are considered as a better mechanism to compare algorithm performance. Section 6.3.2.1 describes the expected target solution for performance profiles. The concept of distance between vectors is discussed in section 6.3.2.2 where a new performance measure is introduced. Extension of the distance measure for larger dimensions is discussed in section 6.3.2.3, whilst advantages of the measure are given in section 6.3.2.4.

6.3.2.1 Unambiguous Comparison Target

Performance measures which produce a vector of performance values have an implicit ordering present within the resulting data vector. The implicit ordering is a result of the optimisation algorithm and the changing problem landscape, and is discarded when calculating aggregated scalars. As a result, due to the significance of the vector value order, the entire vector of results should be considered as the "performance profile" for an optimisation algorithm on a given DOP instance. Result vectors should therefore be compared against other result vectors to determine algorithm performance.

Consider the performance metric result vector $\mathbf{b} = (b_1, b_2, \ldots, b_v)$ that contains n_v performance metric results. Each component within $\mathbf{b}$ represents a measured performance value, sampled for the optimisation at a specific timestep. When $n_v = n_t$, the performance measurement is sampled at the end of every algorithm iteration, whereas the performance measure may be sampled just before the problem landscape undergoes a change with $n_v = n_c$.

Performance measures that are normalised, intuitive and immune to the majority of the shortcomings described in section 6.2.3 are ideal candidates for algorithm performance profiles. One such performance metric is the *relative error* metric (see section 6.2.1.7). The P_{RE} performance measure considers the optimality of solutions in relation to the current problem change period, whilst providing an intuitive representation of the solution quality which is also immune to any landscape scale changes. The one caveat for the P_{RE} performance measure is that the current problem change period optimum is required in order to calculate P_{RE}. P_{RE} has the unique advantage that the ideal target solution is when $P_{\text{RE}} = 1$, thereby defining the target solution vector as $(1, 1, \dots, 1)$. When not possible to use the P_{RE} measure, the P_{WA} measure can be used as an alternative. P_{WA} estimates the target performance vector based on the observed minimum and maximum values of the current set of candidate solutions over a window of algorithm iterations.

6.3.2.2 Distances between Performance Profiles

To determine if one algorithm performance profile is better than another, the performance profile vectors need to be compared. The comparison poses a problem for the analytical process, highlighting the simplicity of using scalar values. Determining the difference between two vectors is not an uncommon task with the differences between binary bit-strings quantified through the use of measures such as the Hamming distance [111]. As a result, the *relative error distance* (P_{RED}) is a metric which can compare the performance profiles of algorithms. Although comparing the difference between different performance profiles is useful, it does not indicate how much *better* a given profile is when compared to others.

The n_v dimensional performance profile $\mathbf{b}$ may be compared to the desired performance profile $\mathbf{d} = (1, 1, \dots, 1)$ to determine the performance profile quality. As $\mathbf{b}$ becomes more similar to $\mathbf{d}$ (*i.e.*, $P_{\text{RED}}(\mathbf{b}) \approx P_{\text{RED}}(\mathbf{d})$), the calculated distance between the vectors decreases. An increase in the dissimilarity of the vectors would indicate performance profiles that are increasingly different. When considering the distance between vectors within a n_v-dimensional Euclidean space, P_{RED} is defined as

$$P_{\text{RED}}(\mathbf{b}) = \frac{\sqrt{\sum_{i=1}^{n_v}(1 - b_i)^2}}{\sqrt{n_v}} \tag{6.21}$$

where n_v is the length of the performance profile and $1 - b_i$ is the difference between the target performance of dimension i to the observed performance b_i.

Figure 6.1 illustrates the distance calculation using equation (6.21) within a 2-dimensional vector space. Consider the example vectors $A = [3, 3, 3, 3]$

and $B = [6, 0, 6, 0]$ that were introduced within section 6.2.3.3. These vectors represent the raw objective function vectors for the P_{BOG} performance measure across 4 algorithm iterations. The vectors A and B can be translated into vectors that contain the relative error by considering the optimisation problem characteristics, which yields $P_{\text{RE}}(A) = [0.5, 0.5, 0.5, 0.5]$ and $P_{\text{RE}}(B) = [0, 1, 0, 1]$. It is now possible to determine that $P_{\text{RED}}(A) = 0.5$ and $P_{\text{RED}}(B) = 0.7071068$ using equation (6.21). From the P_{RED} results, the performance profile of vector A is the preferred result, because the total component-wise distance to the target solution is less than the same component-wise distance for vector B. The performance profile of A presents more consistent component-wise results than that of B, when compared to the ideal problem search space solution.

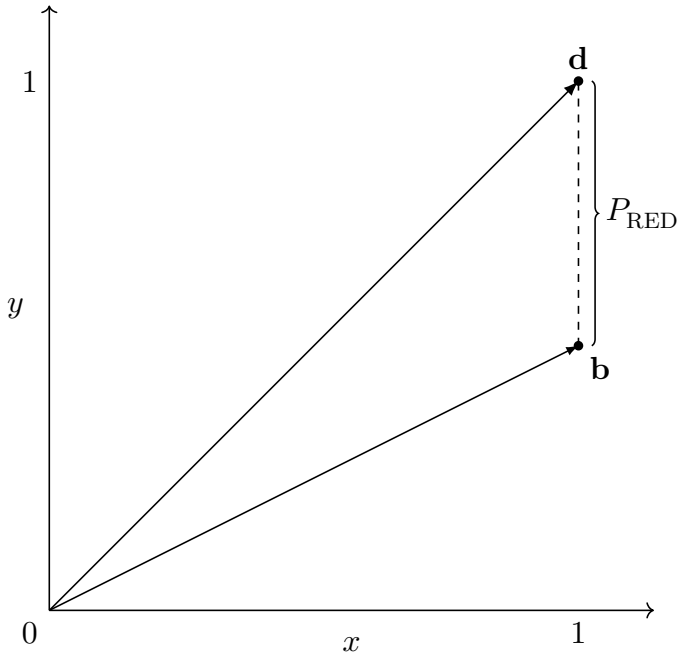


Figure 6.1: P_{RED} calculation within the 2-D Euclidean vector space

6.3.2.3 Relative Error Distances within Large Dimensions

The problems associated with random walk creation within FLA (see section 4.3.1) for high dimensional data as also present for algorithm performance measures when using the Euclidean distance. As the dimensionality of the vector increases, the effects of the curse of dimensionality [144] become more apparent. Beyer *et al.* [14] highlighted this effect, where the distance between high dimensional vectors is shown to no longer be meaningful. Specifically, the results indicated that the ratio between the nearest and farthest elements for different distance measures approach the value of 1 for high dimensional

spaces. As a result, the use of distance metrics to determine proximity in high dimensions produces a result that is effectively meaningless whilst also being unstable, because there is poor discrimination between neighbouring points.

In a study by Aggarwal *et al.* [1], the dimensionality curse is examined in relation to the use of distance metrics for high dimensional data. The L_k norm is examined where the Euclidean distance (the L_2 norm) is shown to be an inferior choice for high dimensional data. Use of the L_1 norm (also known as the Manhattan distance) should instead be preferred for higher dimensional spaces. The study also investigated the use of L_k distance metrics for $k \in (0, 1)$. Such metrics are referred to as *fractional norms* or *quasi-norms* and indicate improved performances in the presence of noisy data. Equation (6.21) is defined using the L_2 norm which remains meaningful for lower dimensions. Within high dimensional spaces L_1 should instead be used for the definition of P_{RED}:

$$P_{\text{RED,L1}} = \frac{\sum_{i=1}^{n_v} (|1 - b_i| + |r_i|)}{n_v} \tag{6.22}$$

where $|1-b_i|$ is the difference between the target and the observed performance b_i for dimension i; r_i is added to the performance difference in order to include the additional distance for component i to fulfil the additional required distance to move to the target value. The vector $\mathbf{r}$ is known as the *rejection vector* for the projection of vector $\mathbf{b}$ onto the target vector $\mathbf{d}$. The rejection vector $\mathbf{r}$ is calculated as the orthogonal vector which satisfies $\mathbf{r} = \mathbf{b} - \mathbf{d}^*$, where $\mathbf{d}^*$ is the vector $\mathbf{b}$ projected onto $\mathbf{d}$. The calculation of the rejection vector is illustrated in figure 6.2 and may be directly calculated from the vectors $\mathbf{b}$ and $\mathbf{d}$ by manipulating the cosine angle formula to produce

$$\mathbf{r} = \mathbf{b} - \frac{\mathbf{b} \cdot \mathbf{d}}{\mathbf{d} \cdot \mathbf{d}} \mathbf{d} \tag{6.23}$$

where $\cdot$ is the dot product between vectors.

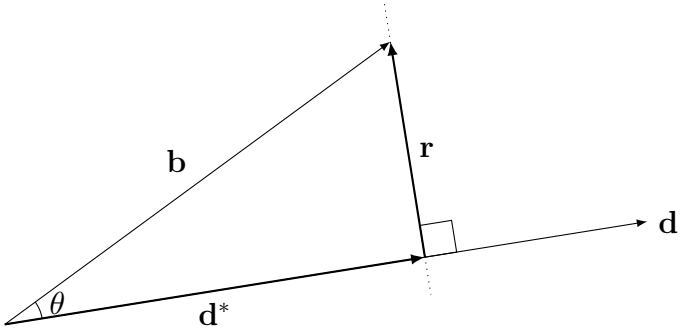


Figure 6.2: Rejection vector calculation

The L_1 metric produces a distance by considering the sum of all vector components. As a result, the distance calculation defined by equation (6.22) is shown in figure 6.3, where the sum of the individual vector components produce a consistent distance metric for high dimensional vectors. If the example vectors A and B are once again considered, we obtain the results of $P_{\text{RED,L1}}(A) = 0.5$ and $P_{\text{RED,L1}}(B) = 1.0$. The results for the $P_{\text{RED,L1}}$ formulation coincide with the results for the P_{RED} measure, whereby the performance profile of vector A is the preferred solution. Vector A contains observed P_{RE} values that are the most similar to the desired target solution vector.

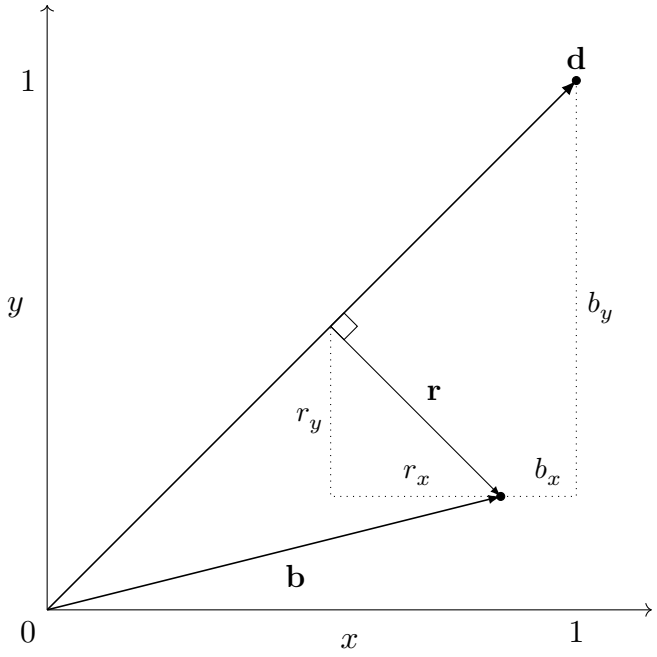


Figure 6.3: $P_{\text{RED,L1}}$ distance calculation as the sum of vector components in a 2-D vector space

6.3.2.4 Advantages of Relative Error Distance

The relative error distance metric has the following advantages that address the shortcomings of DOP performance metrics identified in section 6.2.3. Beneficial characteristics of the P_{RED} metric (including those inherited from the relative error metric) are:

1. **Outlier sensitivity**: The P_{RE} measure places emphasis on the best error observed instead of taking the effect of outliers into consideration. As a result, the P_{RED} measure is also resilient to the effects of outliers and focuses on the error of the best performing candidate solution. Additionally, the relative error does not discriminate against algorithms that

have exploratory behaviour. When a new and possibly better solution is located, the observed relative error will simply reduce.

2. **Flexible sampling strategies**: The number of sampled values which constitute the components of the algorithm's performance profile (*i.e.*, the n_v sampled P_{RE} values) is dependent on the investigation requirements. In the most granular scenario, a P_{RE} value is sampled for every objective function evaluation. Objective function evaluations provide the relative error for each candidate solution to the known best which may be required for a specific study purpose. More generally, the sampled values are focused on the best candidate solution and samples are taken at each iteration. Such sampling will allow for more thorough analysis within DOPs, particularly when considering how the problem changes over time. By taking samples of the performance both before and after the problem landscape change, information about the severity of a change may be determined based on how the candidate solution is affected by the problem landscape change.

3. **Minimal partiality**: The values reported by the P_{RED} measure have a fixed range of $[0, 1]$. Partiality, when considering the metric evaluation process, implies that there are specific cases where the returned metric value is undefined. Undefined values introduce discontinuities within the landscape that would be produced by the performance metric if every point within the problem search space were measured. The P_{RE} metric has an exception where an undefined value may be produced. This only occurs when the problem search space is a flat plane and can easily be accounted for by an inspection of the problem landscape ahead of time or within the implementation of the metric to account for undefined values (such as those created by division with 0).

4. **Intuitive interpretation**: As with the P_{RE} measure the result of P_{RED} is a value within the range of $[0, 1]$, representing the desirability of the solution. Additionally, both the input and the output of the P_{RED} metric calculation has the same domain which adds to the intuition of the measure.

5. **Agnostic optimisation scheme handling**: From the sampled *relative error* values, the notion of good and bad is implicitly encoded into the resulting measure value. Good P_{RE} values are not dependent on the current algorithm optimisation scheme whilst still providing the necessary information to allow for context-free interpretation of the result which is within the range $[0, 1]$. P_{RED} inherits this trait from the P_{RE} measure, albeit in a vector-based form.

6. **Preventing the smoothing effect**: As previously mentioned, the smoothing effect ultimately results in a loss of algorithm performance

information. Aggregated values do not indicate all the nuance that is present within an algorithm's execution and especially not when the underlying optimisation problem changes over time. Each component within the algorithm performance profile is considered when determining the P_{RED} value. As a result, the P_{RED} value provides a more holistic perspective of the algorithm performance. The example vectors A and B from previous discussions have shown how the use of the algorithm performance profile will still provide an indication of which vector has better overall performance when the mean calculation does not. From the algorithm performance profiles, the variance of algorithms across iterations and problem change periods is considered for each component within the n_v-dimensional performance profile.

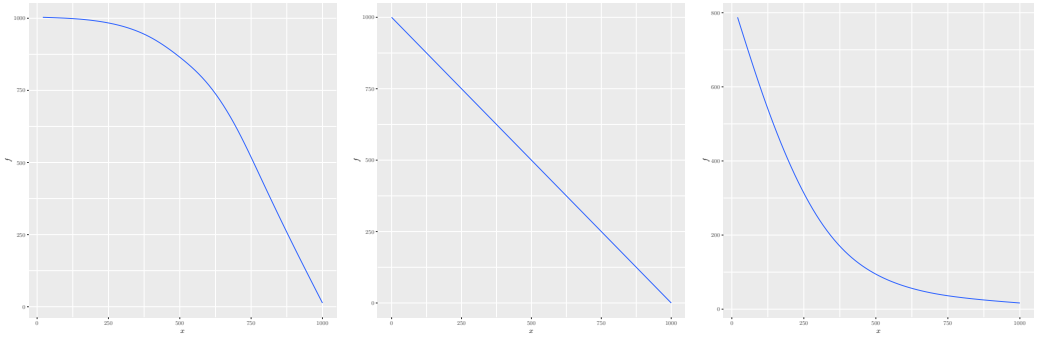
7. **Parameterless construction**: Without additional parameters, P_{RED} can be used to measure algorithm performances without requiring additional computation to tune or configure the metric for use.

8. **Assumption-free data distribution**: No assumptions are made regarding the underlying data distribution which is sampled by the P_{RED} metric. Instead, a bare distance value is calculated from the sampled n_v-dimensional vectors which is an objective view of algorithm performance profiles. The P_{RED} or $P_{\text{RED,L1}}$ measure formulations are available, with the dimensionality of the performance profile determining which measure formulation is most applicable.

6.3.3 Performance Profiles for Static Optimisation Problems

The P_{RED} metric is not limited to use within DOPs, and can also be used within SOPs. Ultimately, the algorithm performance profile is a record of how efficiently an optimisation algorithm is able to obtain the desired solution. Such performance profiles are often represented as plots that track the best objective function value as the optimisation algorithm executes. The measurement frequency is also configurable within SOPs, allowing the measurement to take place per iteration for the best candidate solution, at every objective function evaluation or at some other arbitrarily selected time interval. Using P_{RED} allows for the comparison of these "fitness plots" and will produce a measure of similarity between the actual and observed algorithm performances. Provided that all other aspects of the optimisation problem remain the same and the variation in obtaining solutions is limited to optimisation algorithms only, the performance profiles of the algorithms will be comparable to one another. The benefits previously stated for the use of P_{RED} as a measure of algorithm performance will also apply to static problem search spaces.

Figure 6.4 plots the performance plots of different optimisation algorithms that are searching for a minimum within a static optimisation problem search space. Algorithms $A1$, $A2$ and $A3$ obtain identical final candidate solutions, implying that the objective function values are also the same at the end of 1000 iterations. If the final result were to be used as the overall performance indicator for the algorithms, there would be no observable difference between the algorithms.



(a) Algorithm $A1$ **(b)** Algorithm $A2$ **(c)** Algorithm $A3$

Figure 6.4: Hypothetical algorithm performance profiles for a static optimisation problem

When using algorithm performance profiles instead of the final value, the respective P_{RED} results for algorithms $A1$, $A2$ and $A3$ are 0.7120516, 0.591608 and 0.3698243. From these performance profiles, it can be seen that the best overall P_{RED} value is that of algorithm $A3$. The P_{RED} value indicates that algorithm $A3$ approached the desired optimum value more consistently over the 1000 iterations when compared to the other algorithms. After all, within static problem search spaces, the aim is to obtain the best possible solution as quickly as possible. The more efficiently an algorithm can produce the desired solution implies that an algorithm may obtain the same solution more quickly than other algorithms; that is in fewer algorithm iterations.

6.3.4 Performance Measures and Constrained Optimisation Problems

It has already been established that the P_{RED} measurement is capable of discerning the performance of an optimisation algorithm for both static and dynamic optimisation problem search spaces. Optimisation problem constraints limit the feasible regions of the problem landscape from which candidate solutions can be obtained. For static, constrained optimisation problems the goal of the optimisation algorithm remains the same, that is to obtain the best possible solution within the problem landscape. Although the optimisation

problem constraints place restrictions on the feasible regions within the optimisation problem, it is still possible to determine the optimal solution within the problem search space, even if the calculation is done ahead of time.

Within dynamically changing optimisation problems, the use of optionally changing problem constraints presents a additional computational cost if the best solution within the current problem landscape were to be calculated explicitly. Unfortunately, this is not practical as locating the best solution within the DCOP is one of the goals of the optimisation algorithm, thereby resulting in two optimisation processes executing concurrently. The additional computational cost could only be justified if perfect knowledge of the sample benchmark problems existed, but again this is not generally the case and therefore not practical. The problem is then that within DCOPs, how does the performance of the current optimisation algorithm get quantified when the optimisation problem is constrained and changes over time.

Within DCOPs, where very little in known about the optimisation problem, the optimum value for the optimisation problem search space in the current change period is unknown. Although the precise optimum value is unknown, the following still hold:

- The current performance can still be sampled from the collection of candidate solutions. The bounds of the optimisation problem should also be known, particularly as the bounds information is a requirement for the initialisation process for candidate solutions within the optimisation algorithm. Moreover, the allowable minimum and maximum values for the objective function are usually known, albeit that the values are conservative. The resulting sampled P_{RE} values will represent the current performance of the best candidate solution to the unknown "target" optimum.

- The alterations to the optimisation problem as a result of problem level constraints will serve to either reduce or accentuate features within the problem landscape. Any unknown, conservative bounds should be able to account for the changing feasible and infeasible regions of the problem search space.

- The goal of an optimisation algorith m with DOPs is to locate and to track optima. By allowing the algorithm to continue to find the best possible solutions and to track how these solutions change over time remains unchanged, even when the target optimum is unknown.

- Performance metrics can only measure the candidate solutions within the optimisation algorithm for a given iteration. Information about the characteristics of the optimisation problem may be used within the performance measure, using the conservative values when the problem characteristics are not fully known.

Therefore, the performance profile of the algorithm can be regarded as an estimated performance profile which can be compared to a hypothetical perfect solution, even if such a solution does not exist. The aim is merely to obtain a P_{RED} value that is as close to the hypothetical perfect solution as possible. Furthermore, once the performance profiles are evaluated against the target best solution, the individual performance profiles may then be compared to each other using P_{RED}.

For example, consider the CMPB introduced in section 5.1. The problem constraints are generated through the use of the MPB generator and the final optimisation problem search space is produced as the composition of the constraint and objective function landscapes. The composition specifies that the range of values for the peak heights will be within $[minHeight_g, maxHeight_f]$, but feasible solutions are only found within the range $[0, maxHeight_f]$. As a result, the P_{RE} calculation is still possible, except that the value of $maxHeight_f$ becomes the target optimum value even though none of the peaks within the search space potentially have that value. In other words, the maximal peak value of the objective behaviour space becomes the best possible value that can be achieved for the optimisation problem. From this reasonable assumption, the performance metric sampling may take place and the resulting comparisons between algorithm performance profiles through the use of P_{RED} or $P_{\text{RED,L1}}$ is possible. By using this kind of formulation it is therefore possible to have the optimisation problem search space adjust throughout all problem change periods, whilst comparing performance profiles to a theoretical "best" value. This hypothetical target vector for the performance profile will be the $\mathbf{d} = (1, 1, \ldots, 1)$ vector within the P_{RED} and $P_{\text{RED,L1}}$ calculations.

6.4 Conclusion

This chapter introduced the relevance of performance metrics and how these metrics inform the analysis process. Not all performance metrics exhibit the same behaviour, nor do all metrics produce the same kind of result. Scalar values or vectors may be the produced result from a performance metric. Within the context of DOPs, performance metrics are especially troublesome because the notion of optimality is only relevant between problem change periods. Once the problem landscape undergoes change, performance metrics may not necessarily be compared to each other in a fair manner. If the characteristics of the optimisation problem have changed such that the observed scale within the same metric differs, the comparison is no longer valid. Thankfully, a number of performance metrics exist that take optimisation problem scale changes into account and are able to normalise the performance metric results.

The notion of metric smoothing was discussed, highlighting that the smoothing effect is most problematic when the final metric result is a scalar value. As a result of this smoothing, the metric result may present a partial truth of the

algorithm performance and often necessitates the use of multiple metrics to explain the performance of an algorithm. The smoothing effect is only exacerbated within DOPs and promoted the investigation in a performance metric that uses vectors instead of an aggregated scalar value.

The use of performance vectors, or algorithm performance profiles, provides a detailed view of the algorithm performance at each sampling time interval. The biggest disadvantage of performance vectors is that the vectors are cumbersome to work with even though they represent a more accurate view of algorithm performance. As a result, a new performance metric which determines the distance between performance vectors was introduced. Using this metric (P_{RED}), algorithm performances can be compared such that every component of the performance profile impacts the comparison between two vectors. Therefore, the P_{RED} measure is able to discriminate between algorithm performances by considering much more information that what is available with scalar metrics. Finally, the P_{RED} metric was extended to allow for its use within DCOPs, where there is an uncertainty to the target optimum value within each problem change period.

Part III

Algorithm Performance on Dynamic, Constrainted Optimisation Problems

Chapter 7

Self-Adaptive Quantum Particle Swarm Optimisation

There's a lot of automation that can happen that isn't a replacement of humans but of mind-numbing behavior.

Stewart Butterfield

The QPSO algorithm, as introduced by Blackwell and Branke [17, 18] (see section 3.2.5), describes a PSO variant for dynamic optimisation problems. The problem dependant control parameter r_{cloud} was introduced to control the size of the quantum swarm within the QPSO. The introduction of such a parameter becomes problematic when considering DOPs and DCOPs. It is possible that a portion of the quantum swarm may reside within an infeasible portion of the problem search space, thereby rendering all possible solutions within that part of the quantum cloud as infeasible. The r_{cloud} control parameter should, therefore, become a managed value to ensure that an appropriate radius size is used for the quantum cloud, particularly in DOPs.

This chapter proposes a variant of the QPSO that removes the need to define a value for r_{cloud}, and is organised as follows. Section 7.1 discusses alternative radius strategies that have previously been proposed, with section 7.2 proposing a new radius management strategy that dynamically adjusts the size of r_{cloud} during algorithm execution. The experimental procedure is discussed in section 7.3. The results are analysed in section 7.4, with section 7.5 concluding the chapter.

7.1 Alternative Radius Management Strategies

Blackwell *et al.* [20] suggested using different kinds of probability distributions to influence the movement of quantum particles. Although this approach does hint at adjusting the size of the quantum cloud radius, quantum particles may move to positions beyond the range of the quantum cloud radius, provided that the distribution is a non-bounded distribution. Consider the sampled random values from a multivariate Gaussian distribution; the majority of the sampled random vectors will be within the bounds of the distribution. However, the multivariate Gaussian distribution still allows for sampled points to lie outside of the quantum cloud. Consequently, it is possible to move quantum particles to positions outside the bounds of the problem search space itself. Solutions outside of the problem search space are infeasible solutions and may have boundary constraints applied to them, but at the same time these solutions overcome the problems illustrated and discussed in section 3.2.5.

The performance of QPSO is sensitive to the distribution used. Harrison *et al.* [113] showed that the choice of probability distribution is dependent on the type of search space dynamism. It was also noted that smaller quantum cloud radius values lead to improved QPSO performance, except for chaotic problem landscapes. More importantly, Harrison *et al.* [113] concluded that the uniform distribution is a poor probability distribution choice. In an attempt to remove the r_{cloud} parameter and probability distribution from the QPSO, Harrison *et al.* [112] modified the QPSO to use a PCX [65] operator instead. The result of the modification is an algorithm which cannot be regarded as a QPSO variant because the quantum metaphor no longer exists within the algorithm. However, the resulting algorithm does address the problem with diversity loss, because the PCX operator was designed to introduce diversity. Unfortunately, the modified algorithm also introduces two additional control parameters, namely the deviations of Gaussian distributions used within the PCX operator itself.

7.2 Self-Adaptive Quantum Particle Swarm Optimisation

An alteration to the QPSO algorithm is proposed in this section which dynamically adapts the value of the quantum cloud radius, r_{cloud}. The self-adaption process of r_{cloud} considers the neutral and quantum particle subgroups within the QPSO independently. The diversity for each subgroup is calculated by using particles that are contained within feasible regions of the problem search

space, defined as

$$D(S(t)) = \frac{1}{n_s} \sum_{i=1}^{n_s} \sqrt{\sum_{j=1}^{n_x} (x_{ij}(t) - \bar{x}_j(t))^2} \tag{7.1}$$

where n_s is the size of the particle swarm and $\bar{x}_j(t)$ is the average value for the j-th dimension across all feasible particles. The average value for dimension j, is calculated as

$$\bar{x}_j(t) = \frac{\sum_{i=1}^{n_s} x_{ij}(t)}{n_s} \tag{7.2}$$

The initial value for r_{cloud} is calculated by considering the initial positions of particles within the problem search space. The maximum diversity is considered based on the effect of changes within the DOP instance. When the problem search space undergoes a change, the location of problem optima before the change may no longer be present in the newly changed problem search space. A larger search radius may facilitate locating new, possibly better, optima by considering more of the problem search space. Figure 7.1 presents an artificial scenario of a QPSO swarm: Given a number of algorithm iterations, without the problem search space undergoing any changes, the particle swarm starts to converge on the global best particle, $\hat{\mathbf{y}}(t)$. As the swarm converges onto the global best particle, the size of r_{cloud} (*i.e.*, the diversity of the swarm) also reduces in size. After a change to the problem search space, the optimum value moves from $\hat{\mathbf{y}}(t)$ to a new location in the problem search space (with label A) and the particle swarm no longer contains any problem optima. The change in the problem search space causes the size of r_{cloud} to once again increase by having quantum particles re-initialise within the problem search space domain. The quantum particle re-initialisation facilitates exploration within the new problem search space by increasing the diversity of the quantum particles, which results in an increase in the size of r_{cloud}. The iteration process proceeds to continue, with the size of r_{cloud} once again reducing as the swarm converges onto a new search space position.

The rate of r_{cloud} size reduction is determined by the type of probability distribution used in equation (3.6). A bounded distribution places a limit on the upper bound value that the distribution may return. Consequently, the upper bound limit also defines the upper bound value that any dimension within the position of the quantum particle may assume. Over a number of iterations, the resulting trend will be to constrain quantum particles around the global best particle more aggressively as r_{cloud} continues to reduce in size. Unbounded distributions allow sampling values that are larger than the defined r_{cloud} value. Sampling values that are larger than r_{cloud} will resist collapsing onto the global best particle for longer than a bounded distribution. Figures 7.2 and 7.3 illustrate a sample of 1000 randomly selected points within a radius of 1 from the origin. The upper bound value is visible in figure 7.3 with no sample points extending past the defined boundaries.

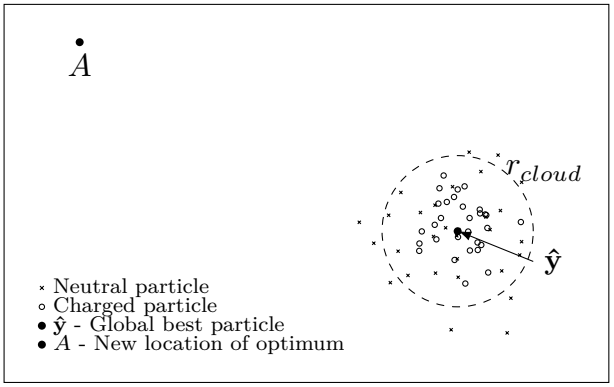


Figure 7.1: Artificial scenario after problem landscape experiences change with high spatial severity

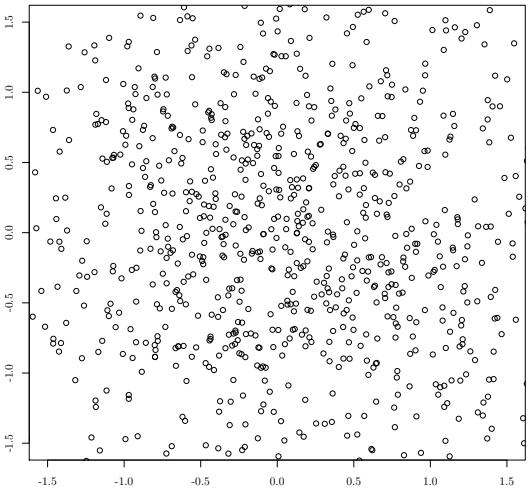


Figure 7.2: Multi-variate Gaussian distribution in two dimensions

The r_{cloud} value is calculated as

$$r_{cloud} = \max\{D(S_n(t)), D(S_q(t))\} \tag{7.3}$$

where $S_n(t)$ and $S_q(t)$ are the neutral and quantum particle subgroups at time step t, with the diversity of each particle group obtained using equation (7.1). Figure 7.4 illustrates the calculation of the r_{cloud} value and how this value defines the radius of the quantum cloud hyper-sphere, within which quantum particles move, centred at the current global best particle.

The memory of the neutral particles may be stale after a problem search space change occurs within the DOP, referring to positions that are no longer best positions. The memory of the neutral particles should update to match the new problem search space, otherwise neutral particles may be attracted

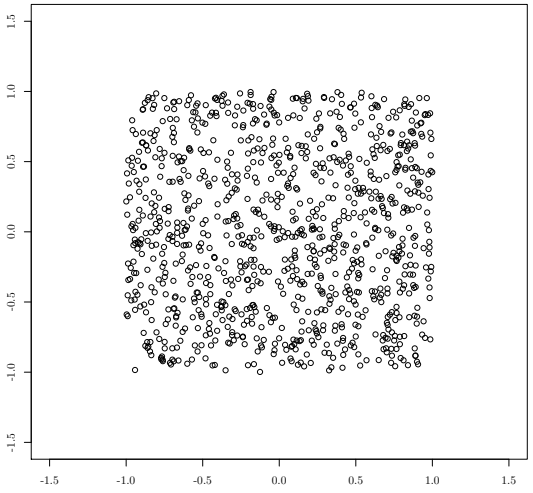


Figure 7.3: Two-dimensional multi-variate uniform distribution

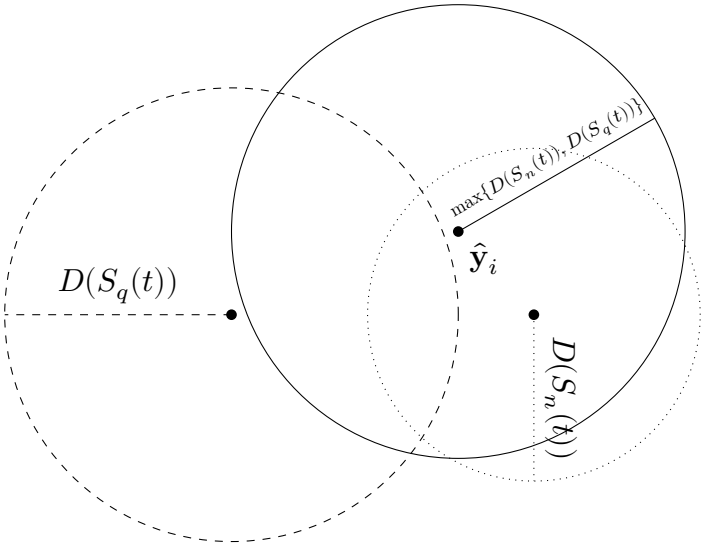


Figure 7.4: Dynamic radius calculation from subgroup diversity values

to areas of the search space which are now undesirable. The personal best position of neutral particles updates only if the new best position remains within the problem search space [86]. Algorithm 7.1 provides pseudo-code for the SaQPSO algorithm.

7.3 Experimental Procedure

The main objective of this experimental work is to demonstrate that a self-adapting r_{cloud} is either better than or performs similar, when compared against

Algorithm 7.1 Self-adaptive Quantum Particle Swarm Optimisation

$(S_n, S_q) \leftarrow$ Initialise swarm with subgroups for neutral and quantum particles
$r_{cloud} \leftarrow$ Calculate quantum radius using equation (7.3)
repeat
  for each neutral particle $p \in S_n$ **do**
    update velocity using equation (3.1)
    update particle position using equation (3.2)
    calculate and assign objective function value
    update the personal best position **iff** the position is within problem domain
  end for
  for each quantum particle $p \in S_q$ **do**
    update position using equation (3.6)
    calculate and assign objective function value
  end for
  $r_{cloud} \leftarrow$ Calculate quantum radius using equation (7.3)
until *stopping condition is true*

control algorithms with predefined values for r_{cloud}. In other words, no significant difference in performance should exist between QPSOs with static or adapting r_{cloud} values.

This section describes the required considerations to allow for the evaluation of the QPSO and the SaQPSO, on a set of dynamic optimisation problems. Section 7.3.1 describes the design of the experiments, with section 7.3.2 describing the measurements which quantify algorithm performance, together with the statistical process.

7.3.1 Experimental Design

The most important consideration of the experimental work is that the underlying problem search spaces are exactly the same between algorithm executions. Without the guarantee that the same problem is used for different algorithms, a fair comparison of algorithm performance is not possible. Furthermore, execution of QPSO variants should not produce side-effects as the algorithm executes that could alter the optimisation problem. If the optimisation problems were changed during the execution of the algorithm, it would not be possible to perform a fair comparison between algorithms unless it could be guaranteed that the same changes are performed by the other algorithms. To ensure that this property remains an invariant for all experiments, the software library CIlib was developed to allow for the precise control over optimisation algorithms and optimisation problems. CIlib, a crucial part of the experimentation process, is

not relevant for this discussion and is instead presented within part IV of this thesis.

PSO parameter choices are based on PSO convergence properties [41, 115, 285], with the *lbest* topology providing slower information propagation throughout the neutral particles. Sampling a Gaussian distribution centred at the global best position of the neutral particle subgroup allows for quantum particle movement with a central tendency at the global best position. Quantum particle movement from the global best position of the neutral particle subgroup still allows for unconstrained movement that may exceed the boundaries of the quantum cloud. Table 7.1 lists the PSO control parameter values for the QPSO algorithms. For the experimental work, three QPSO algorithms with static $r_{cloud} \in \{5, 10, 50\}$, together with the SaQPSO are considered. The static QPSO variants are identified by the size of the associated radius, namely QPSO-5, QPSO-10, and QPSO-50 for r_{cloud} values of 5, 10 and 50 respectively.

Table 7.1: QPSO algorithm parameters

Parameter	Value
Particles	40
Proportion quantum particles	50%
ω	0.729844
c_1, c_2	1.496180
Topology	l-best (size 3)
Iteration strategy	Synchronous
PRNG Seed	123456789L
Static radius values	$r_{cloud} \in [5, 10, 50]$
Quantum cloud distribution	Gaussian

The benchmark problems were defined to match the classification of Duhain and Engelbrecht [77], using the MPB generator (refer to section 4.1.3) to create the problem instances. Each problem search space contained 10 peaks and was configured using the parameters defined in table 7.2. Each problem search space classification was also defined to be a Type III problem search space [123]. Each algorithm configuration was executed for a total of 30 independent executions, for 1000 iterations. The optimisation problem search space defined a five dimensional landscape, with each dimension bound to the domain $[0, 100]$.

7.3.2 Performance Measures for Quantum Particle Swarm Optimisation Algorithms

Duhain [76] and Duhain and Engelbrecht [77] recommend that better choices for performance measurement of dynamic problem search spaces include the

Table 7.2: MPB benchmark generator parameters

Parameter	Static	Progressive	Abrupt	Chaotic
Peak count	10	10	10	10
Peak height	[30, 70]	[30, 70]	[30, 70]	[30, 70]
Peak width	[1, 12]	[1, 12]	[1, 12]	[1, 12]
Height change severity	1	1	10	10
Width change severity	0.05	0.05	0.05	0.05
Change severity	1	1	10	10
Random movement % (λ)	0	0	0	0
Change freqency (iterations)	200	1	200	5

accuracy of the solutions over time, the *stability* (solution quality after the problem search space experiences change), and algorithm *exploitative capacity*, which is the quality of the best solution between the changes of a problem search space. Whilst considering the performance measures used for the QPSO in literature [18, 20, 76], a subset of the measurements are selected. This subset will allow for simpler comparisons to QPSO results within existing literature. As a result, vector-based performance measurements are not considered for these experiments. The set of "good" performance measures which observe the performance of all QPSO algorithms include

- CME (see section 6.2.1.3),
- ABEBC (see section 6.2.2.2), and
- ABEAC (see section 6.2.2.3).

The performance results of the QPSO algorithms was tested by applying a Mann-Whitney-U rank sum test, together with a Holm correction in order to determine if there is a significant difference ($\alpha = 0.05$) in algorithm performance. For each comparison between algorithms, a value of 1 is allocated to the superior algorithm performance with the inferior performance being allocated a value of -1. In the case where there is no difference between algorithm performances (*i.e.*, a tie in performance) a value of 0 is allocated to both algorithms. The allocated scores are then summed to produce a *wins-minus-losses* aggregate.

7.4 Experimental Result Analysis

This section contains the analysis of the experimental results for the four algorithms (QPSO-5, QPSO-10, QPSO-50 and SaQPSO). Sections 7.4.1 to 7.4.3

respectively discuss the results for the CME, ABEBC and ABEAC performance measurements. An analysis of the size of the quantum cloud follows in section 7.4.4.

7.4.1 Analysis of Collective Mean Error

Table 7.3 provides algorithm rankings with respect to the CME measurement. For the CME measurement, the rankings indicate that the SaQPSO performed the best across the different problems search space types. QPSO-50 was the second best performing algorithm, followed by QPSO-10 and QPSO-5. As shown in figure 7.5, a similar trend to the ranking data can be observed when comparing algorithm performances. For the abrupt and progressive problem spaces all algorithms achieved similar performances, but the wins-minus-losses favours the SaQPSO within these problem search spaces.

After the problem search space experiences a change, the SaQPSO has an increase in diversity, as illustrated in figure 7.8. The increase in diversity results in a larger area for quantum particles to explore, and as the swarm starts to converge on an optimum, the radius value decreases. With a decreasing radius, quantum particles begin exploitation of the search space around the optimum. The error values in figure 7.5 also show that the QPSO is sensitive to the frequency of problem search space change: the lower error values were achieved for the quasi-static search space where the frequency of change is low. Compared to the static QPSOs, it should be noted that the SaQPSO did not perform worse.

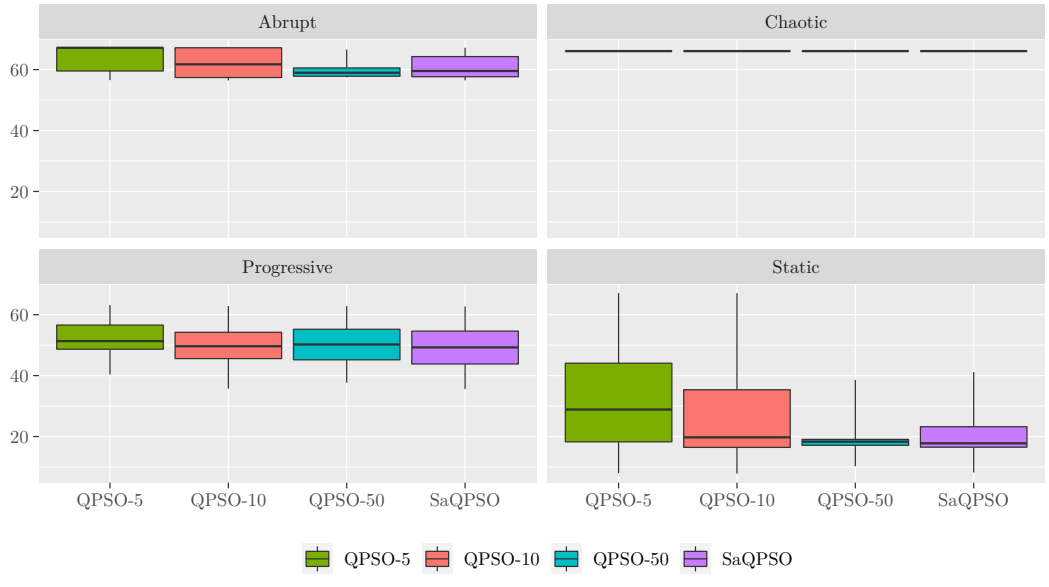


Figure 7.5: CME performance results over MPB problem search space types

Table 7.3: Algorithm performance ranking

Problem	Measure	QPSO-5	QPSO-10	QPSO-50	SaQPSO
Quasi-static	CME (Win/Loss)	(0/-3)	(1/-2)	(2/-1)	(3/0)
	ABEBC (Win/Loss)	(0/-3)	(1/-2)	(3/0)	(2/-1)
	ABEAC (Win/Loss)	(0/-3)	(3/0)	(1/-2)	(2/-1)
	Win+Loss	-9	-1	3	5
	Rank	1	2	3	4
Progressive	CME (Win/Loss)	(0/-2)	(2/-1)	(0/-1)	(2/0)
	ABEBC (Win/Loss)	(0/-2)	(2/-1)	(0/-1)	(2/0)
	ABEAC (Win/Loss)	(0/-2)	(2/-1)	(0/-1)	(2/0)
	Win+Loss	-6	3	-3	6
	Rank	1	3	2	4
Abrupt	CME (Win/Loss)	(0/-3)	(1/-2)	(2/0)	(2/0)
	ABEBC (Win/Loss)	(0/-3)	(1/-2)	(2/-1)	(3/0)
	ABEAC (Win/Loss)	(0/-3)	(1/-2)	(2/-1)	(3/0)
	Win+Loss	-9	-3	4	8
	Rank	1	2	3	4
Chaotic	CME (Win/Loss)	(0/-3)	(1/-2)	(2/-1)	(3/0)
	ABEBC (Win/Loss)	(0/-3)	(1/-2)	(2/-1)	(3/0)
	ABEAC (Win/Loss)	(0/-3)	(1/-2)	(2/-1)	(3/0)
	Win+Loss	-9	-3	3	9
	Rank	1	2	3	4
Win/Loss total		-33	1	7	28

7.4.2 Analysis of Average Best Error Before Change

Figure 7.6 illustrates that all four algorithms managed to achieve values of less than 20 for the ABEBC within the quasi-static problem search space. For the other search spaces, the same trend of the CME measurement is evident, with none of the algorithms particularly providing a clearly better solution, and a similar spread of error values. Because the ABEBC demonstrates the exploratory capacity of an algorithm, it is clear that none of the algorithms were able to effectively locate a new solution before the search space experienced change. The SaQPSO achieved comparable performance when compared to the static QPSO variants.

7.4.3 Analysis of Average Best Error After Change

After the problem search space experiences change, the QPSO-10 and SaQPSO managed to achieve median values that are lower than that of the other QPSO algorithms for the quasi-static problem spaces. Unfortunately, for the other problem search space types, no one algorithm displayed a clear improvement,

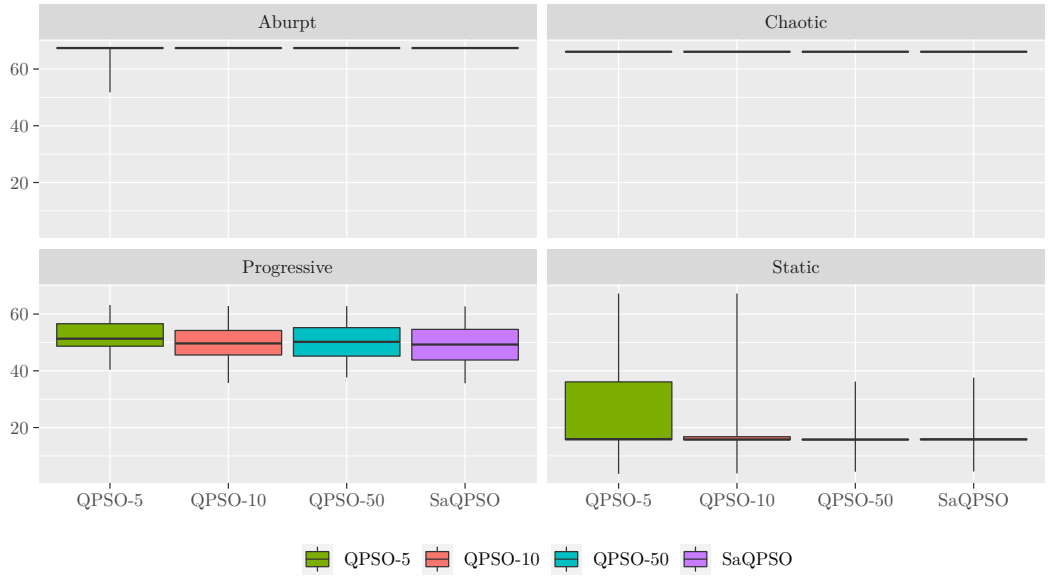


Figure 7.6: ABEBC performance results over MPB problem search space types

and all algorithms (including the SaQPSO) achieved equally poor results. These performances are illustrated in figure 7.7.

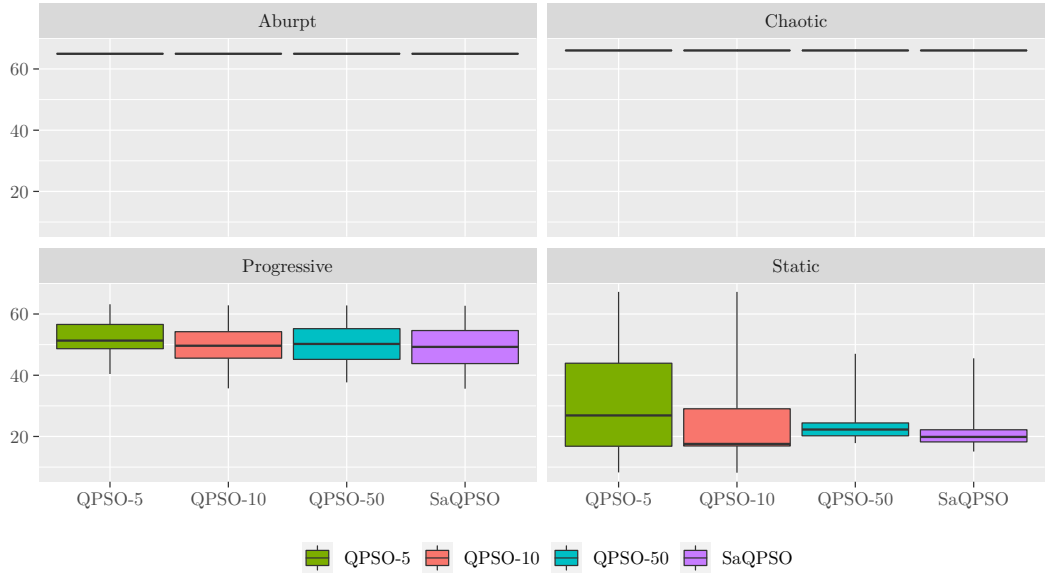


Figure 7.7: ABEAC performance results over MPB problem search space types

7.4.4 Analysis of the Dynamic Quantum Radius and Diversity

For the SaQPSO, the average radius size is illustrated in figure 7.8 for each problem search space type over 1000 iterations. From the graph it is clear that the diversity (which is the cloud radius value), did change over the course of algorithm execution. For search spaces with high temporal severity (chaotic and progressive), the cloud radius size fluctuated at a large value which is roughly half of the problem domain. Due to the frequency of the search space changes, there is not enough time between the problem space changes for particles to share enough information in order to attract the swarm to a specific region within the search space. Therefore, the re-initialization process maintains a large diversity.

The quasi-static problem search space plot shows that the radius reduced to a small value and increased as the problem search space changed (every 200 iterations), albeit a small change. The size of the cloud radius for the abruptly changing problem spaces reduced similarly to the quasi-static problems, but at 400 iterations, increased to a value under half of the problem domain size and remained there for the remainder of the algorithm execution. It is not clear why this behaviour is observed. As expected, the size of the cloud radius remained large for the progressive and chaotic problem search spaces where the frequency of change is high.

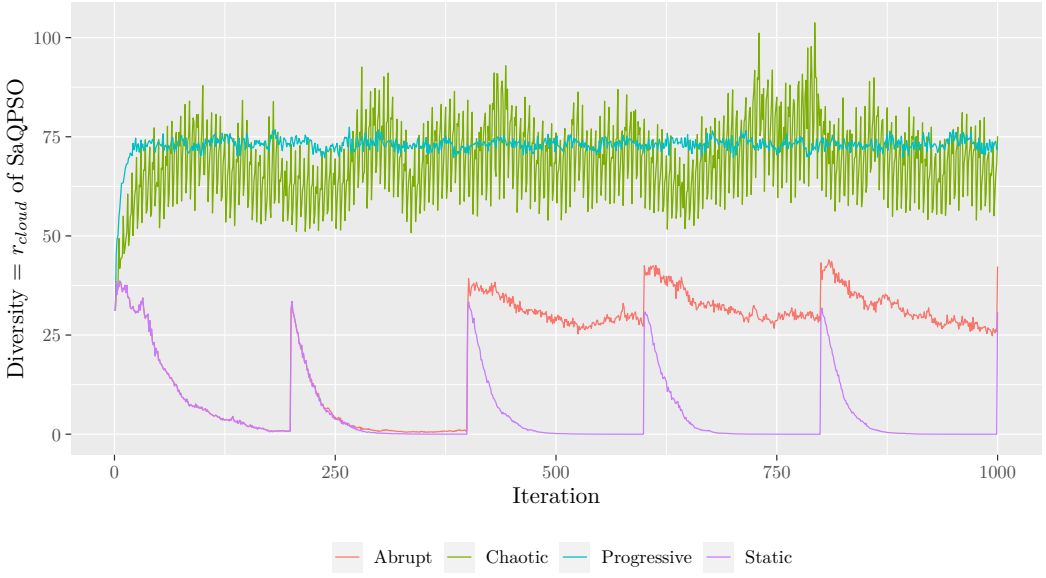


Figure 7.8: Average diversity/quantum cloud radius across algorithm iterations for all MPB problem search space types

7.5 Conclusion

This chapter investigated the radius management of the QPSO. During the investigation it was observed that the quantum cloud radius parameter, r_{cloud}, is a problematic algorithm control parameter that is problem dependant. Alternative radius management strategies have been proposed, however, the results do not always leave the behaviour of the QPSO algorithm unchanged. A new variant of the QPSO for dynamic problem search spaces was proposed where the value of r_{cloud} can adjust dynamically during the execution of the algorithm, thereby removing the requirement to determine a suitable value for r_{cloud} in advance.

From the reported experimental results and the analysis of performance measurements, it was shown that the proposed SaQPSO algorithm did not perform any worse, nor any better than three statically configured QPSO algorithms. The results indicate a promising outcome: the SaQPSO may substitute any QPSO algorithm without the loss of performance whilst providing the benefit of letting the r_{cloud} control parameter become independent of the optimisation problem. The next chapter incorporates the SaQPSO into an algorithm suitable for DCOP.

Chapter 8

Co-Evolutionary Algorithms for Dynamic, Constrained Optimisation Problems

If I have seen further it is by standing on the shoulders of Giants.

Isaac Newton

Dynamic optimisation algorithms have been shown to be successful in both tracking and maintaining solutions within a changing optimisation problem landscape. When optimisation problem constraints are considered in addition to the dynamic problem landscape, the use of simpler constraint handling methods (such as penalty functions) are preferred. Penalty functions are particularly favourable due to the minimal additional complexity they introduce into the optimisation process, whilst being simple to implement. When considering DCOPs, problem landscapes become particularly challenging for optimisation algorithms as the complexity of the optimisation problem increases. A co-evolutionary approach decomposes the more complex problem search spaces into an alternative representation. Within this alternative representation, constraint violations are minimised in a separate process simultaneous to minimisation of the objective function. The general co-evolutionary framework (discussed in section 3.3) provides an overview of the co-evolutionary process for both cooperative and competitive framework variants.

In this chapter, a new dynamic co-evolutionary framework is proposed to cater for dynamic, constrained optimisation problems. The new framework formulation takes inspiration from the cooperative co-evolutionary particle swarm optimisation (CCPSO) algorithm [256], but differs by having the inner individual optimisation algorithms be dynamic variants of static optimisation algorithms. Notably, this new formulation allows for the definition of dynamic co-evolutionary algorithms to solve DCOPs instances.

This chapter is structured as follows: Section 8.1 discusses the reformulation of constrained optimisation problems, using a co-evolutionary approach. The proposed dynamic co-evolutionary framework is proposed in section 8.2, together with a discussion of possible algorithms to solve DCOPs. The chapter concludes with section 8.3.

8.1 A Co-Evolutionary Approach for Static, Constrained Optimisation Problems

Shi and Krohling [256] proposed a formulation of the co-evolutionary framework using two PSO swarms. This algorithm formulation, hereafter referred to as CCPSO, transforms the optimisation problem into an unconstrained min-max problem, through the use of Lagrangian multipliers (see definition 2.5). This is in contrast to the more commonplace approaches which involve the preservation of feasible solutions, the repair of infeasible solutions, the use of decoders, penalty functions, and hybrid algorithms. Figure 8.1 illustrates the how the PSOs cooperate within the CCPSO.

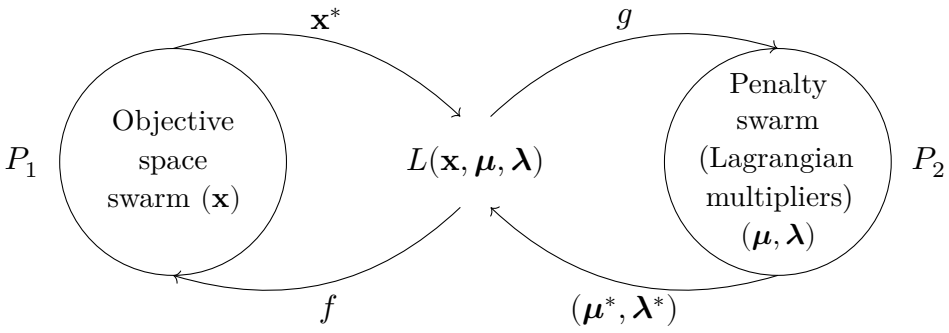


Figure 8.1: Graphical representation of the CCPSO knowledge transfer and PSO specific objective function creation between cooperating PSOs.

Within figure 8.1, the dual Lagrangian formulation $L(\mathbf{x}, \boldsymbol{\mu}, \boldsymbol{\lambda})$ is the mechanism through which the swarms are able to cooperate. Each swarm obtains its fitness function by filling the missing parameters within $L(\mathbf{x}, \boldsymbol{\mu}, \boldsymbol{\lambda})$ with information obtained from the other cooperating swarm. The fitness function f of the objective space swarm (*i.e.*, the "min" swarm) is obtained by partially applying $L(\mathbf{x}, \boldsymbol{\mu}, \boldsymbol{\lambda})$ with the current best Lagrangian multipliers, $(\boldsymbol{\mu}^*, \boldsymbol{\lambda}^*)$, from the penalty swarm. The resulting fitness function for the objective space swarm is

$$f(\boldsymbol{\mu}^*, \boldsymbol{\lambda}^*) = \min_{\mathbf{x} \in P_1} L(\mathbf{x}, \boldsymbol{\mu}^*, \boldsymbol{\lambda}^*) \tag{8.1}$$

where $\boldsymbol{\mu}^*$ and $\boldsymbol{\lambda}^*$ respectively represent the coefficients to the inequality and equality constraints of $L(\mathbf{x}, \boldsymbol{\mu}, \boldsymbol{\lambda})$ and P_1 represents the particles of the objective space swarm.

For the penalty swarm (*i.e.*, the "max" swarm), the fitness function g represents the penalty function within $L(\mathbf{x}, \boldsymbol{\mu}, \boldsymbol{\lambda})$ and is obtained by partially applying $L(\mathbf{x}, \boldsymbol{\mu}, \boldsymbol{\lambda})$ with the current best solution, $\mathbf{x}^*$, within objective space. The fitness function for the penalty swarm is defined as

$$g(\mathbf{x}^*) = \max_{\boldsymbol{\mu}, \boldsymbol{\lambda} \in P_2} L(\mathbf{x}^*, \boldsymbol{\mu}, \boldsymbol{\lambda}) \tag{8.2}$$

where P_2 represents the penalty swarm particles.

At each iteration of the CCPSO the fitness function of each swarm is updated before each swarm updates its particles using the velocity update (see equation (3.1)) and position update (see equation (3.2)) of the canonical PSO algorithm (see algorithm 3.3). Pseudo-code for the CCPSO algorithm is provided in algorithm 8.1.

Algorithm 8.1 Co-Evolutionary Particle Swarm Optimisation

$L(\mathbf{x}, \boldsymbol{\mu}, \boldsymbol{\lambda}) \leftarrow$ Formulate the Lagrangian optimisation problem (*i.e.*, the dual problem) ▷ *definition 2.5*
$P_1 \leftarrow$ Initialise PSO swarm for solution
$P_2 \leftarrow$ Initialise PSO swarm for Lagrangian multipliers
repeat
  $g(\boldsymbol{\mu}, \boldsymbol{\lambda}) \leftarrow$ Apply current best solution from P_1 to $L(\mathbf{x}, \boldsymbol{\mu}, \boldsymbol{\lambda})$
  $f(\mathbf{x}) \leftarrow$ Apply current best solution from P_2 to $L(\mathbf{x}, \boldsymbol{\mu}, \boldsymbol{\lambda})$
  Execute PSO P_1 with objective function f ▷ *Algorithm 3.3*
  Execute PSO P_2 with objective function g ▷ *Algorithm 3.3*
  Re-evaluate *personal best* positions for particles in P_1 and P_2
until stopping condition(s) satisfied

8.2 Co-Evolutionary Approach for Dynamic, Constrained Optimisation Problems

This section proposes an approach to solve DCOPs using a co-evolutionary algorithm formulation, which is similar to the algorithm formulation in section 8.1. When considering DCOPs, an optimisation algorithm should be able to not only adapt to the changing problem landscape (by tracking and maintaining solutions), but should also adapt to any changing problem constraints.

A new dynamic co-evolutionary algorithm framework is presented in section 8.2.1, whilst section 8.2.2 discusses possible algorithm combinations that use the dynamic co-evolutionary framework.

8.2.1 Proposed Co-Evolutionary Framework for Dynamic, Constrained Optimisation Problems

For each DCOP category, the following observations can be made based on the complexity of the problem search space and actions required when either the problem landscape or constraints change:

- SOSC problem search spaces are the most simplistic DCOP problem class. The CCPSO described in section 8.1 is sufficient to provide solutions, because both problem search space and constraints remain static.
- SODC problems provide a challenge to the optimisation algorithm due to the constraints changing over time. The method of constraint handling should adapt in order to cater for the changes to the optimisation problem constraints.
- DOSC problem spaces are somewhat similar to SODC in that only a single aspect of the optimisation problem experiences change. The problem constraints remain static whilst the problem landscape experiences changes. As a result, a simpler constraint handling method may be sufficient, whilst a dynamic optimisation algorithm can optimise the objective problem space.
- DODC problem spaces are the most complex combination, where both the problem landscape and the problem constraints change. As a result, it is reasonable that the optimisation algorithm should be able to adapt to changes to both the objective landscape and the optimisation problem constraints, during the optimisation process.

The co-evolutionary approach described in section 8.1 allows for both the min function and the max function to be simultaneously optimised. Unfortunately, for dynamically changing optimisation problems the CCPSO is not capable of responding appropriately to any changes experienced by the optimisation problem. As a result, the particles from both the objective space swarm and the penalty swarm will suffer from outdated memory values (see section 3.2) where the best found positions are no longer valid. Furthermore, the particles from both the objective space swarm and the penalty swarm will also lose diversity as both swarms settle onto an equilibrium state.

When the optimisation problem constraints change, the number of problem constraints need not remain the same. If the number of constraints changes, then so too does the number of Lagrangian multipliers. Therefore, the search space dimensionality of the CCPSO penalty swarm changes and necessitates that the particle position and the particle velocity vectors also update to match the new search space dimensionality.

By replacing the static optimisation algorithms (*i.e.*, the PSOs) within the CCPSO with dynamic optimisation algorithms, the co-evolutionary approach

Algorithm 8.2 Dynamic Co-Evolutionary Framework

$L(\mathbf{x}, \boldsymbol{\mu}, \boldsymbol{\lambda}) \leftarrow$ Formulate the Lagrangian optimisation problem (*i.e.*, the dual problem) ▷ *definition 2.5*
$P_1 \leftarrow$ Initialise dynamic optimisation algorithm for objective space solutions
$P_2 \leftarrow$ Initialise dynamic optimisation algorithm for Lagrangian multipliers
repeat
  Detect and react to objective space and/or problem constraint changes for P_1 and P_2 candidate solutions
  $g(\boldsymbol{\mu}, \boldsymbol{\lambda}) \leftarrow$ Apply current best solution from P_1 to $L(\mathbf{x}, \boldsymbol{\mu}, \boldsymbol{\lambda})$
  $f(\mathbf{x}) \leftarrow$ Apply current best solution from P_2 to $L(\mathbf{x}, \boldsymbol{\mu}, \boldsymbol{\lambda})$
  Execute dynamic optimisation algorithm with objective function f on candidate solutions from P_1
  Execute dynamic optimisation algorithm with objective function g on candidate solutions from P_2
until stopping condition(s) satisfied

may be adapted for DCOPs. The result is a co-evolutionary algorithm framework that uses dynamic optimisation algorithms to simultaneously optimise both the objective function and the constraint handling function for DCOP instances. Optimisation of these functions then provides a solution to the DCOP itself through the cooperation of the dynamic optimisation algorithms. Additionally, the proposed dynamic co-evolutionary framework can be applied to all DCOP categories, including SOSC problem instances irrespective of no changes being experienced by the SOSC optimisation problem. A notable aspect of the dynamic co-evolutionary framework is that any change within the optimisation problem search space or in the problem constraints produces a new optimisation problem search space. This effect is also demonstrated for the CMPB in chapter 5.

Algorithm 8.2 provides high-level pseudo-code for the proposed dynamic co-evolutionary framework for DCOPs.

8.2.2 Proposed Co-Evolutionary Algorithms

By using the proposed dynamic co-evolutionary framework, it is possible to define different co-evolutionary algorithms for DCOPs. When examining the Lagrangian transformation, it is evident that the "domains" of the dynamic optimisation algorithms are not the same. The Lagrangian multiplier vector (the combination of both μ and λ multiplier coefficients) does not maintain the same decision variable domain as that of the solution vector, and changes as the number of constraints within the optimisation problems changes. When selecting dynamic optimisation algorithms to optimise the candidate solutions of either P_1 or P_2, the domains of the candidate solutions should be considered.

For example, assume that the same dynamic optimisation algorithm is used for both the objective space solutions and the Lagrangian multipliers. As noted in section 9.3.2, the tuning of dynamic optimisation algorithm control parameters does not make sense. However, the algorithmic control parameters still control the ultimate behaviour of the optimisation algorithm. When algorithmic control parameters have values that are either too large or too small, the performance of the dynamic optimisation algorithm will be affected negatively. Therefore, dynamic optimisation algorithms with fewer user defined control parameters, or with self-adapting control parameters, are better choices within the dynamic co-evolutionary algorithm framework.

Six DCOP optimisation algorithms are proposed using the dynamic co-evolutionary framework together with dynamic optimisation algorithms. The dynamic optimisation algorithms that are considered for use with the dynamic co-evolutionary framework manage the diversity of the candidate solutions by either introducing diversity or maintaining diversity levels. The same dynamic optimisation algorithm is used for both the objective and Lagrangian multiplier algorithms. The proposed dynamic co-evolutionary algorithms all contain dynamic optimisation algorithms that do not require any problem-specific control parameters, with the same dynamic optimisation algorithm used for both P_1 and P_2 candidate solutions. Changes to the optimisation problem are detected by candidate solutions observing a decrease in fitness. When the optimisation problem experiences a change in either the objective space or in the constraint space, the candidate solutions are re-evaluated to update any maintained memory values. If a change in dimensionality is observed for the optimisation problem constraints, the candidate solutions of P_2 are re-initialised. The proposed algorithms, based on the dynamic co-evolutionary framework are

CCRIGA: The RIGA allowed the GA to be applied to DOPs by introducing new genetic material, which is randomly generated after each algorithm iteration. These "random immigrants" ensure that the diversity of the candidate solutions remain high from iteration to iteration. Embedding RIGA into the dynamic co-evolutionary framework allows for the definition of the cooperative co-evolutionary random immigrant genetic algorithm (CCRIGA).

CCHyperM: Hyper-mutation adapts the GA by observing the diversity within the candidate solutions, allowing for a period of "hyper mutation" in order to improve candidate solution diversity. The resulting co-evolutionary algorithm, using the internal HyperM for both P_1 and P_2 candidate solutions, is known as the cooperative co-evolutionary hyper mutation genetic algorithm (CCHyperM).

CCSaDE: SaDE is an adaptive algorithm, whereby the control parameters of the algorithm are adapted in order to obtain the best parameter

combination for the current optimisation problem. The diversity of the SaDE is maintained through one of the defined DE trial vector creation processes within the algorithm. Using the SaDE as the individual optimisation algorithms within the dynamic co-evolutionary framework defines the cooperative co-evolutionary self-adaptive differential evolution (CCSaDE) algorithm.

CCSaQPSO: As previously mentioned, the SaQPSO algorithm is a self-adaptive PSO variant for dynamic environments, without the requirement to specify the problem-dependent control parameter r_{cloud}. SaQPSO attempts to maintain the diversity of the candidate solutions throughout the execution of the algorithm. The dynamic co-evolutionary algorithm that uses the SaQPSO is known as cooperative co-evolutionary self-adaptive quantum paritcle swarm optimisation (CCSaQPSO).

CCGVPSO: GVPSO defines a dynamic PSO variant that moves particles throughout the problem search space by controlling the particle step-size with a Gaussian distribution. Although GVPSO was originally developed for SOPs, it is possible to use GVPSO within a DOP because the movement of the particles is resilient to landscape changes. Particle diversity is maintained due to the probabilistic particle movement described in equations (3.4) and (3.5).The dynamic co-evolutionary variant of the GVPSO is the cooperative co-evolutionary gaussian-valued particle swarm optimisation (CCGVPSO) algorithm.

CCSaQGVPSO: Within SaQPSO, the neutral particles adhere to the update process of the canonical PSO algorithm (*i.e.*, equations (3.1) and (3.2)). The self-adaptive quantum gaussian-valued particle swarm optimisation (SaQGVPSO) algorithm is identical to the SaQPSO algorithm, except that the neutral particles use the update equations of the GVPSO instead. SaQGVPSO is another dynamic variant of PSO, which can be used within the proposed dynamic co-evolutionary framework to produce the cooperative co-evolutionary self-adaptive quantum gaussian-valued particle swarm optimisation (CCSaQGVPSO) algorithm.

8.3 Conclusion

This chapter discussed the CCPSO algorithm for constrained, static optimisation problems. From the static co-evolutionary algorithm framework a new dynamic co-evolutionary framework was derived and proposed. The new co-evolutionary framework substitutes dynamic variants of the static optimisation algorithms within the co-evolutionary framework to allow for use within dynamic, constrained optimisation problems. Using the Lagrangian method for constraint handling, it is recommended that self-adaptive dynamic optimisation

algorithms are used within the dynamic co-evolutionary framework, allowing for consideration of the different candidate solution domains. Six dynamic co-evolutionary algorithms were discussed and proposed as co-evolutionary algorithms for DCOPs, using different approaches to manage algorithm diversity. These newly proposed dynamic co-evolutionary algorithms will be used within the empirical work in subsequent chapters.

Chapter 9

Empirical Process

Observation is a passive science, experimentation an active science.

Claude Bernard

Optimisation algorithms operating within DOP problem landscapes are handicapped: Although the ultimate goal is to locate and to track solutions within the problem search space, the manner in which the underlying optimisation problem changes is unknown to the algorithm. Furthermore, the different kinds of changes that a problem search space may experience is not finite even though the types of changes can be classified into distinct categories. When considering changing constraints within a problem search space, the complexity present within a changing landscape increases even further, providing a great challenge to the optimisation process and algorithm. Using the dynamic co-evolutionary framework and the derived algorithms from section 8.2.2, experiments and simulations can be defined in order to test the effectiveness of algorithms, defined within the dynamic co-evolutionary framework, to solve DCOPs. For comparison, non-coevolutionary algorithms are also considered to solve DCOPs.

The experimental procedure followed to test the dynamic co-evolutionary optimisation algorithms, as well as the non-coevolutionary optimisation algorithms, is presented in this chapter and is structured as follows: Section 9.1 presents the research questions for the experimental work, together with their motivation. The benchmark problems for the simulations are discussed in section 9.2, whilst section 9.3 discusses the selection of algorithms for the simulations in addition to the defined dynamic co-evolutionary algorithms. In order to test the efficacy of the optimisation algorithms on the benchmark problems, the selection of performance measurements are discussed in section 9.4. Section 9.5 presents the statistical procedure with which the algorithm performances are compared. This chapter concludes in section 9.6.

9.1 Motivation for Experimental Work

A fair synopsis of the research into DCOP problem instances would be that the current work has only started to scratch the surface. DCOPs provide an incredible challenge to optimisation algorithms, where the amount of variability within the problems is potentially greater than the capability of current optimisation algorithms. Furthermore, existing studies on solving DCOPs [83, 166, 171, 179, 207, 245, 311] have focused on benchmark problems that are comparatively simple, maintaining only a few decision variables. Nevertheless, a better understanding of algorithm behaviour and performance on such problems may allow for the design and development of algorithms that are better suited to adapting to the variability within DCOPs. For a comprehensive set of DCOP problem instances, the performance characteristics of the proposed dynamic co-evolutionary algorithms (defined in section 8.2.2) are compared to a set of non-coevolutionary dynamic optimisation algorithms. The following research questions are formulated for the experimental work in order to compare these optimisation algorithms:

1. *Can algorithms based on the dynamic co-evolutionary framework provide solutions to DCOPs?*

   It has already been established that CCPSO is capable of providing solutions to SOSC problem instances [256]. Is it reasonable to expect that the use of dynamic optimisation algorithms within a co-evolutionary algorithm will result in feasible solutions to DCOPs? Furthermore, is it reasonable to expect that feasible solutions are obtained for even the most complex category of DCOPs? This research question is addressed by evaluating the six newly proposed dynamic co-evolutionary algorithms (see chapter 8), the SaQPSO (see chapter 7) and five non-coevolutionary algorithms on a comprehensive set of DCOP instances.

2. *Does the choice of constraint handling approach matter?*

   The dynamic co-evolutionary framework transforms the optimisation problem through the use of the Lagrangian transformation. As a result, the Lagrangian multipliers are optimised during the execution of the algorithm and adapt as dictated by the inter-algorithm knowledge transfer and objective function evaluation. Alternative constraint handling methods, such as the addition of penalties, have been shown to be effective for SOSC optimisation problems. Penalty based constraint handling may be applied to more complex categories of DCOP, provided that the penalty functions can adapt to the changing optimisation problem. Even so, it is not certain that penalty based constraint handling be preferred over the co-evolutionary Lagrangian formulation.

   This research question is addressed by evaluating the performance of the same *base optimisation algorithm*. The base optimisation algorithm

is a specific dynamic optimisation algorithm (*e.g.*, RIGA) from which variant optimisation algorithms are derived. From the base optimisation algorithm, pairs of optimisation algorithm variants are compared which either use the Lagrangian transformation or the α-constraint approach to handle optimisation problem constraints.

3. *Does the ratio of feasible to infeasible solutions reduce as the problem complexity increases?*

   Less complex optimisation problem landscapes should allow an optimisation algorithm to more effectively focus on solutions within feasible problem space. As a result, the ratio of feasible to infeasible solutions should favour feasible solutions. If the ratio instead favours infeasible problem spaces, the implication would be that the algorithm is unable to guide the search process, or that the algorithm is simply exploring within infeasible search space regions.

   This research question is addressed by evaluating the percentage of feasible solutions produced by an optimisation algorithm across all algorithm iterations. These vectors of feasibility are then compared using the P_{RED} measure to determine how effectively the optimisation algorithm can produce feasible solutions when compared to the hypothetical best, which consists of only feasible solutions.

4. *Does candidate solution diversity provide an indication of algorithm performance for DCOPs?*

   The importance of candidate solution diversity is established with dynamic optimisation algorithms. Enough diversity ensures that current solutions continue to be tracked and that new solutions can be located within the optimisation problem search space.

   Given the complexity of DCOPs, it is assumed that candidate solution diversity is an important consideration for optimisation algorithms because of the changes these problem instances can experience. Furthermore, do the constraint handling methods within these optimisation algorithms impact the candidate solution diversity during the execution of the optimisation algorithms?

   This research question is addressed by evaluating the candidate solution diversity across all algorithm iterations, for all objective space and constraint space behaviours.

5. *Which approaches recover the best after the optimisation problem experiences a change?*

   Once the optimisation problem experiences a change, the current optimisation algorithm is required to adapt in order to continue to track

solutions and to locate new feasible solutions within the DCOP. An algorithm with a good recovery ability should be able to notice the change in the optimisation problem. Once the change is identified, the optimisation algorithm can adjust its control parameters, the candidate solutions or a combination of both in order to continue tracking and locating feasible solutions. The Lagrangian formulation of the optimisation problem presents as a more complex optimisation process, necessitating a co-evolutionary algorithm to simultaneously optimise the objective decision variables and penalty function. This is in contrast to the complexity of the optimisation process when a single population based, dynamic optimisation algorithm with an adaptive penalty based constraint handling method is used. Would the recovery process of the optimisation algorithm be influenced by the choice of constraint handling method?

This research question is addressed by evaluating the performance of the dynamic co-evolutionary algorithms and the non-coevolutionary algorithms before, after and across the changes experienced by DCOP instances.

6. *Are there differences in optimisation algorithm performance based on the category of DCOP instance?*

   DCOPs are the composition of both the objective landscape and the constraint landscape. One possible categorisation of DCOPs differentiates instances based on the dynamic nature of the objective and constraint landscapes. From this categorisation, the possible instance groups are SOSC, SODC, DOSC and DODC problem instances.

   Another possible categorisation is based on the behaviour of the changes experienced by the objective space and the constraint space. For each constituent landscape of the final composed optimisation problem, the landscape behaviour can be categorised based on the spatial and temporal severity of the search space changes. This allows for the grouping of problem instances based on progressive, abrupt, chaotic or static change behaviour for both the objective landscape and the constraint landscape.

   From these categorisations, are any differences observed for the performance of the optimisation algorithms? This research question is addressed by evaluating the performance profiles of the optimisation algorithms for above mentioned categories of optimisation problem instances.

9.2 Benchmark Problem Instances

As descried in chapter 5, it is possible to define a comprehensive set of DCOP benchmark problems by building on the classification of Duhain and Engelbrecht [77]. The CMPB problem generator can produce DCOP instances by

composing a MPB objective problem space together with a MPB constraint problem space. As a result, a total of $28^2 = 784$ unique problem instances can be generated with the CMPB generator. These problem instances differ in complexity and range from static objective and constraint landscapes, to landscapes that combine chaotic objective and chaotic constraint landscapes. Following from this definition of the possible problem search spaces, all of the DCOP categories (*i.e.*, SOSC, SODC, DOSC and DODC) are represented within the resulting set of benchmark problems.

The objective and constraint landscape MPB generators have the same benchmark function generator parameters and change frequencies, provided in table 9.1. As mentioned within section 5.1, each problem instance is uniquely labelled by considering the problem space behaviour of both the objective and constraint landscapes. For example, when considering the benchmark problem label `A1R/C1L`, the objective space behaviour experiences abrupt landscape changes with random, type I optima movement. In contrast, the constraint landscape experiences chaotic landscape changes, together with linear, type I movement of the optima.

The dimensions defined within the CMPB parameters are slightly larger than the dimensions used within currently available literature, as highlighted in section 4.2, providing optimisation problem landscapes that are more complex. Although the CMPB may easily be defined for large dimensions, the selected dimension is low enough to hopefully still provide a similar intuition about the problem spaces, when considering other studies. Furthermore, high dimensional search spaces introduce other complications [1, 144, 172, 181, 283, 315] (such as the curse of dimensionality, interdependence of dimensions and high dimensional smoothing), and are not considered within this study.

Table 9.1: MPB benchmark generator parameters

Parameter	Static	Progressive	Abrupt	Chaotic
Domain	[0, 100]	[0, 100]	[0, 100]	[0, 100]
Number of dimensions	5	5	5	5
Peak count	10	10	10	10
Peak height	[30, 70]	[30, 70]	[30, 70]	[30, 70]
Peak width	[1, 12]	[1, 12]	[1, 12]	[1, 12]
Height change severity	1	1	10	10
Width change severity	0.05	0.05	0.05	0.05
Change severity	1	1	10	10
Random movement % (λ)	0	0	0	0
Change frequency (iterations)	∞	20	100	30

9.3 Algorithms and Control Parameters

The sections that follow discuss the set of optimisation algorithms for DCOPs together with the control parameters of these algorithms. Section 9.3.1 discusses both newly proposed and existing optimisation algorithms for DCOPs, together with the assigned constraint handling method for the algorithms. The algorithm control parameters for the discussed algorithms are presented in section 9.3.2.

9.3.1 Algorithms

Together with the six proposed dynamic co-evolutionary algorithms of section 8.2.2 and SaQPSO (see chapter 7), an additional five dynamic algorithms are considered for the experimental work. These additional algorithms have been taken from existing literature to determine if these algorithms are capable of providing solutions to DCOPs, simply with the addition of a constraint handling method. The algorithms taken from literature are

- RIGA [104]
- HyperM [43]
- DDECV [2]
- SaDE [238]
- GVPSO [115]

These DOP algorithms have been shown to provide good performance and are able to maintain decision space diversity. Diversity is managed by either maintaining the current, already present levels of diversity, or by introducing new diversity into the algorithm during execution.

The proposed dynamic co-evolutionary algorithms use the Lagrangian transformation to convert the DCOP into an unconstrained DOP, handling constraints as part of the optimisation process. For the additional DOP algorithms and SaQPSO, constraint handling is achieved through the use of the α-constraint [271, 273] which provides an adaptive penalty function. Table 9.2 summaries the algorithms based on type and constraint handling method used within the experimental work. Note that SaQPSO, CCRIGA, CCHyperM, CCSaDE, CCSaQPSO, CCGVPSO and CCSaQGVPSO are all new algorithms proposed by this thesis.

9.3.2 Algorithm Control Parameter Selection

Emphasis to obtain an optimal set of control parameters for optimisation algorithms already exists, particularly within static optimisation problem search

Table 9.2: Optimisation algorithm and constraint handling association

Constraint handing	Algorithm type	Algorithm
α-constraint	Dynamic	RIGA
		HyperM
	Adaptive	SaQPSO
		DDECV
		SaDE
		GVPSO
Lagrangian transformation	Co-evolutionary	CCRIGA
		CCHyperM
		CCSaDE
		CCSaQPSO
		CCGVPSO
		CCSaQGVPSO

spaces. The procedure to obtain such a parameter set often requires the consideration of all possible parameter value combinations, which grow exponentially as more control parameters are considered. Tools [172] have been developed in order to reduce the complexity of combinatorial searches of the control parameter space. A statistical approach is taken within the tools to determine when a control parameter change will not yield better results, upon which large portions of the control parameter search space can be excluded. The general idea is that, for a given optimisation problem, each algorithm should be compared using its own best set of control parameters. Doing so will give each algorithm the best possible chance to provide the best solution to the common optimisation problem.

Within dynamically changing problem landscapes, the number of algorithm control parameter sets would need to increase to match the number of landscape change periods. Due to a static problem landscape existing between landscape changes, it is conceivable that an optimal set of algorithm control parameters for each landscape change period should be determined because the landscape characteristics may have changed. Unfortunately, such a strategy is simply not practical with the currently available tools. As a result, tuning optimisation algorithm control parameters for DOPs can be seen as a fruitless endeavour. Currently, algorithm control parameters are not managed for each change period and tuned control parameters are usually only valid for the first problem landscape. There is no guarantee that the selected control parameter values are optimal for subsequent problem landscapes [6, 22, 113, 156, 212, 283], possibly becoming ineffective as the landscape changes. This will remain the

status quo until experimental frameworks that explicitly control randomness and optimisation algorithm control parameters become more commonplace within the research community.

Due to the above mentioned concerns, the choice of optimisation control parameters is based on previous literature where appropriate. For example, PSOs have the advantage of preexisting theoretical work [40, 41, 234, 278] which defines the criteria for "good" control parameters. Where such information is not available, the preference is for self-adaptive control parameter strategies which adjust and adapt the algorithm control parameters over time. Initialisation of all algorithms is done uniformly. Particles within the PSOs are initialised to have zeroed velocity vectors, with personal best updates only occurring when improved solutions are found within the bounds of the problem search space. Table 9.3 lists the control parameters for the mentioned algorithms.

Table 9.3: Selected optimisation algorithm control parameters

Algorithm	Parameter	Value	Source
General	n_s	50	
RIGA	p_c	0.1	[104, 209]
	p_m	0.15	
	p_{im}	0.3	
HyperM	p_c	0.6	[43, 209]
	p_m	0.001	
	p_{hyper}	0.5	
DDECV	p_c	0.6	[2]
	F_{before}	0.9644	
	F_{after}	1.0820	
QPSO	Quantum split	50%	[18, 112, 113, 156]
	ω	0.729844	
	c_1, c_2	1.496180	
	Topology	l-best (size 3) + quantum particles	
	Iteration strategy	Synchronous	
GVPSO	e	0.5	[115]

9.4 Performance Measures

The performance measures attempt to quantify how efficiently and accurately the optimisation algorithms produce solutions. The following performance measures were observed for each optimisation algorithm, at every algorithm iteration:

Accuracy To determine the accuracy of an optimisation algorithm, the error of the best solution is recorded at each algorithm iteration, producing the algorithm performance profile (see section 6.3). Performance profiles provide a unique "signature" for the performance of a particular optimisation algorithm on a given problem instance. Importantly, the optimisation problem instance should be identical for all optimisation algorithms, thereby eliminating differences in optimisation algorithm performance resulting from differences in the optimisation problems. The proposed P_{RED} measurement (see section 6.3.2) determines the similarity between a given performance profile to the hypothetical ideal solution performance profile. Different performance profiles can then be compared based on the value of the P_{RED} measure. The P_{RED} measure will be the main measurement for algorithm accuracy across optimisation problem search spaces.

Diversity The convergence between candidate solutions can be measured by the diversity of the candidate solutions. Lower diversity values indicate an increase in the homogeneity of the candidate solutions. It has already been shown that dynamic optimisation algorithms benefit from larger diversity values because the goal of these algorithms is to track current and locate new optima within the changing problem landscape. Therefore, it is also important for a DCOP optimisation algorithm to maintain the correct levels of diversity to not only continue to track and locate new solutions, but to guide the search for solutions within feasible regions of the problem landscape. Equation (7.1) provides the diversity calculation for this performance measure.

Recovery The ability of a dynamic optimisation algorithm to recover after the problem landscape experiences change allows the algorithm to continue tracking and searching for new candidate solutions. As with the diversity measurement, this characteristic of DCOP optimisation algorithms is important in order to understand if the algorithm can recover to feasible solutions. After a change to the optimisation problem, current optima may have not only moved to new search space locations, but likely into infeasible regions of the changed problem search space. To determine how effective the DCOP optimisation algorithm's recovery process is, three different measurements are considered:

- ABEBC (see section 6.2.2.2) provides the indication of the average error of the candidate solutions before the problem search space undergoes a change.
- The maintained error after the problem search space change is determined by the ABEAC measurement (see section 6.2.2.3).
- The rate at which the optimisation algorithm recovers to a solution is given by the ARR measurement (see section 6.2.2.5).

Solution Feasibility This measure indicates the percentage of the candidate solutions that are located within the feasible regions of the optimisation problem search space. It should also be noted that different diversity introduction and maintenance methods may produce candidate solutions within infeasible regions of the problem search space. Even so, the feasibility percentage still provides an indication of which feasibility region most of the candidate solutions occupy.

From these measures, multiple measurement vectors are produced. These measurement vectors contain the iteration-based chronological behaviour of the optimistion algorithm across the change periods of the optimisation problem. Section 6.3 discusses the reasons for preferring measurement vectors instead of scalar values, because they provide for a more holistic view of an optimisation algorithm for a given DOP instance. When considering DCOPs, the importance of such vector-based measurements becomes even more important based on the number and type changes experienced by the optimisation problem or problem constraints. The number of change periods within DCOPs increases as both optimisation problem landscapes and problem constraints change.

9.5 Statistical Analysis Process

To determine if optimisation algorithms are able to effectively locate, maintain and refine solutions within the different combinations of objective and constraint spaces, the optimisation algorithms are evaluated on all 784 CMPB problem instances (refer to section 9.2). For each CMPB problem instance, each algorithm was evaluated for 50 independent executions, where each independent execution contained 1000 algorithm iterations.

For performance measures that produced a vector of measurements, the performance profiles for the each dependant execution were compared using the P_{RED} measure (refer to section 6.3.2.2). These resulting samples of P_{RED} results were then used as the input for analysis in the statistical procedure outlined by García and Herrera [96] and García *et al.* [97]. For each benchmark problem a Friedman test was performed to determine if any algorithm produced a result that was significantly different from the other algorithms. If the Friedman test indicated that at least one of the algorithms produced a significant result, a

post-hoc test was then performed with Schaffer's correction [96, 97] applied to the resulting p-values. A significance level of $\alpha = 0.05$ was assigned for all the performed statistical tests.

To prevent the smoothing concerns of aggregate values, whilst not considering the underlying distribution of the results, the median performance from each of the 50 algorithm executions was taken as the representative performance of each algorithm in order to assign wins and losses. If the reported performance was better than the other performances, the associated algorithm was assigned a win (value of 1), whilst the inferior performances were assigned a loss (value of -1). In the case of ties between algorithms, a value of 0 is assigned to each of the algorithms. From these wins and losses, the difference in performance (also known as the win-minus-losses) was calculated and then used to determine the final ranking of the algorithm performance, for each benchmark problem. The statistical procedure is illustrated in figure 9.1 and implemented using the `scmamp` [30] library for `R` [274].

The overall performance of the optimisation algorithms may be illustrated with a *critical difference plot*. A critical difference plot displays the critical range together with the calculated test-statistic value for each optimisation algorithm. Any optimisation algorithm towards the left hand side of the test-statistic axis has achieved a better performance than the optimisation algorithms immediately to the right. Optimisation algorithms that score a test-statistic value contained within the critical difference range, have achieved a statistically significant result. When any optimisation algorithms are connected by a horizontal line, beneath the value test-statistic axis, these linked optimisation algorithms display equivalent performance statistically. For such linked optimisation algorithms, the left most algorithm is still the preferred optimisation algorithm choice.

In order to address the concerns of reproducibility which were highlighted in previous chapters, the CI software library CIlib [222, 224, 226] was used to implement, execute and collect the data of the performed experiments. As a result, the experiments can be fully reproduced and the description of how CIlib enables this important property is discussed and motivated within part IV of this thesis. For all experiments, the initial positions of the candidate solutions and all problem landscape changes were identical. Therefore, the obtained execution results allow for fair comparisons by limiting any variability observed within the results to be solely that of the optimisation algorithm.

9.6 Conclusion

This chapter proposed the evaluation procedure for dynamic co-evolutionary algorithms and dynamic optimisation with the α-constraint penalty method, on a comprehensive set of benchmark problem instances. A set of research questtions were proposed to determine the efficacy of the optimisation algo-

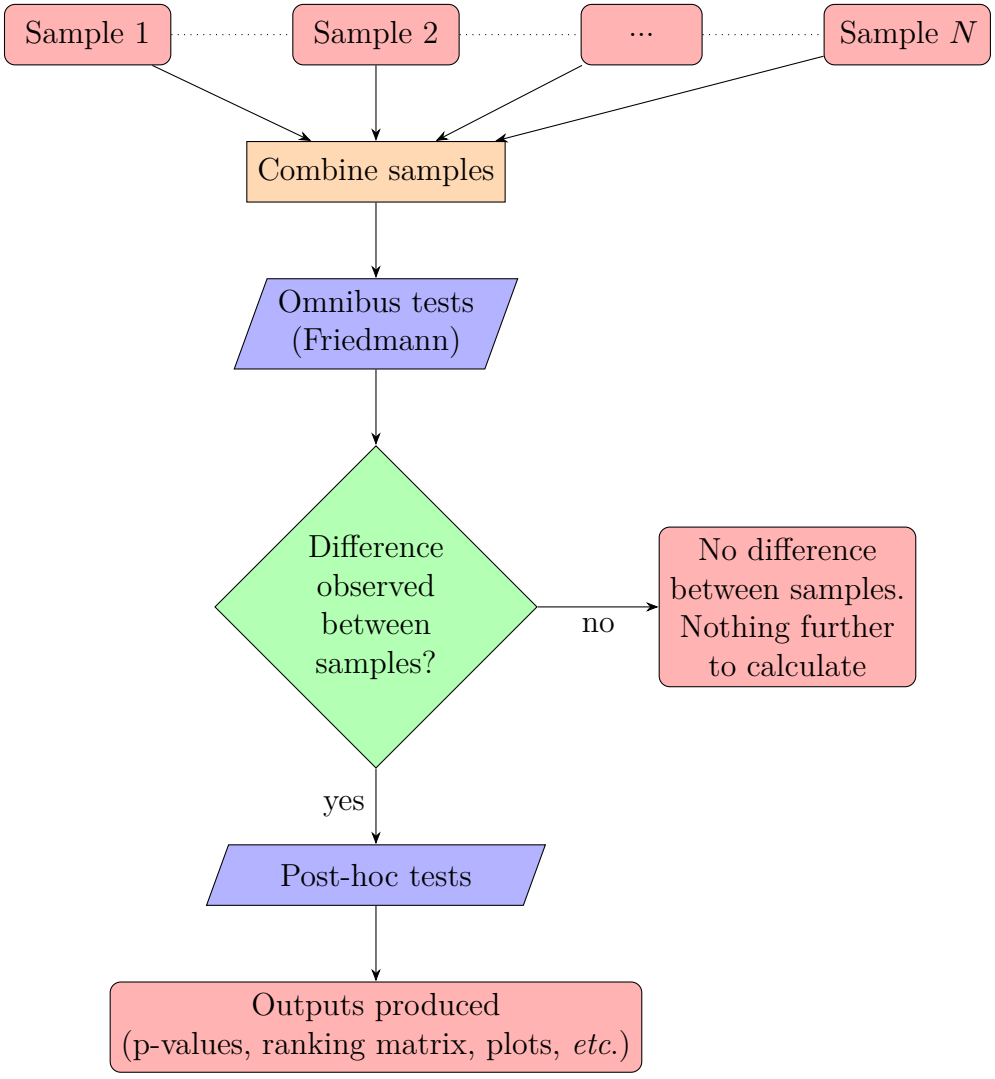


Figure 9.1: Statistical procedure for the comparison of algorithms

rithms using a set of measurements for the empirical work. In order to obtain answers to the research questions the statistical procedure was discussed. This statistical procedure follows established guidelines for the evaluation of multiple optimisation algorithms. The next chapter will use this procedure to compare the performances of the optimisation algorithms.

Chapter 10

Performance Analysis

Science is beautiful when it makes simple explanations of phenomena or connections between different observations. Examples include the double helix in biology and the fundamental equations of physics.

Stephen Hawking

Although the ultimate goal of an optimisation algorithm is to locate and to track solutions within the problem search space, the manner in which the underlying optimisation problem changes is unknown to the algorithm. Moreover, the different kinds of changes that a problem search space may experience is not finite even though the types of changes can be classified into distinct categories. When considering changing constraints within a problem search space, the complexity present within a changing landscape increases even further, providing a great challenge to the optimisation process and algorithm.

Chapter 9 presented the optimisation algorithms for the experimental work. This chapter aims to determine how effective these DCOPs optimisation algorithms are at providing solutions to DCOPs, whilst providing answers to the research questions that are enumerated in section 9.1.

This chapter presents the analysis of the results obtained from the experimental work. Section 10.1 presents the analysis of the DCOP optimisation algorithm accuracy. The maintained diversity of the optimisation algorithms is presented in section 10.2, whilst optimisation algorithm recovery is discussed in section 10.3. Optimisation algorithm solution feasibility is discussed in section 10.4, before conclusions for the chapter are presented within section 10.5.

10.1 Analysis of Algorithm Accuracy

Optimisation algorithm performance profiles indicate how effectively the algorithm is able to provide solutions to an optimisation problem. Specifically, the

performance profiles represent the accuracy of the optimisation algorithm as the underlying problem instance changes over time. In order to investigate the performance of algorithms, the results in this section are analysed from two perspectives of the DCOP instance, namely the objective space behaviour perspective and the constraint space behaviour perspective. These perspectives are derived from the *objective space behaviour/constraint space behaviour* label of each individual problem instance.

Consider the constraint space behaviour `P2L`. For the objective space behaviour perspective of the `P2L` constraint space behaviour, all algorithm performance samples are considered where the constraint space behaviour is `P2L`. This provides an indication of how optimisation algorithms performed across all objective space behaviours, with a fixed constraint space behaviour. This is repeated for the remaining constraint space behaviours to provide an indication of algorithm performance with changing objective space behaviours. A similar process for the constraint behaviour space perspective is done by fixing the objective space behaviour and then considering all algorithm performance samples. The performance samples of the optimisation algorithms are vectors of fitness error values, quantified using the P_{RED} measurement.

This section presents the algorithm performances from the perspective of the objective space behaviours in section 10.1.1. The constraint space behaviours perspective is given in section 10.1.2, before the analysis of optimisation algorithm accuracy in section 10.1.3.3. The performance of the optimisation algorithms on a sample of simple, moderate and complex DCOPs optimisation problem instances is presented in section 10.1.4. This section concludes with a summary in section 10.1.5.

10.1.1 Objective Space Behaviour

The purpose of this section is to analyse the ability to cope with objective space changes whilst the constraint behaviour space is fixed. Figures 10.1 and 10.2 illustrate the objective space results across all optimisation algorithms as a series of box-plots. Across all the constraint space behaviours, the same trend in optimisation algorithm performance is visible. The only algorithms that consistently provide the smallest inter-quartile range are CCSaDE and DDECV, albeit that some outliers are observed. This shows that the algorithms provide more consistent results, compared to the other algorithms, especially when considering that a P_{RED} value that is closer to 0 indicates greater result accuracy.

If the optimisation algorithms were to be paired up based on the base algorithm they were derived from, similar performance results become visible for these algorithm pairs. These optimisation algorithm pairings consist of both the dynamic co-evolutionary algorithm and dynamic optimisation algorithm variant. The algorithm pairings of CCRIGA and RIGA, CCHyperM and HyperM, CCGVPSO and GVPSO, as well as CCSaQPSO and SaQPSO display

similar reported inter-quartile ranges, with differing median results. Generally, the median result for CCRIGA is better than that of RIGA, with the opposite being observed for CCHyperM and HyperM. Only within the progressive and static constraint behaviour spaces is a shift observed between CCSaQPSO and SaQPSO. Although the median result comparison between CCGVPSO and GVPSO appears to be indistinguishable, CCGVPSO does provide slightly better median results for the majority of the objective spaces. Within these constraint space behaviours, the median result for SaQPSO becomes noticeably worse than that of CCSaQPSO.

For the algorithms that indicate larger variances within their results, namely CCRIGA, RIGA, CCSaQPSO and SaQPSO, median results are close to that of the lower-bound of the inter-quartile range. Again, the only exception is for SaQPSO on the progressive and static constraint space behaviours. Larger inter-quartile ranges are also observed for SaDE for all constraint space behaviours, except for the progressive and chaotic constraint behaviour spaces. With an increased number of changes for these behaviour spaces, the variance within the inter-quartile range is small, but the reported P_{RED} values are larger than those reported for most of the other algorithms. An interesting observation is that the performance difference between CCSaDE and SaDE is notably large, with CCSaDE providing superior performances when compared to SaDE. Within the `STA` objective behaviour space, a similar result is present even though no changes are experienced with the optimisation problem search space.

From these results, the overall behaviour of CCSaDE and DDECV does indicate that the algorithms are able to adjust to, and cope with, changes with respect to the fixed objective behaviour space. This does not mean that the other algorithms are not able to provide solutions, but the results are not as desirable as that of CCSaDE and DDECV.

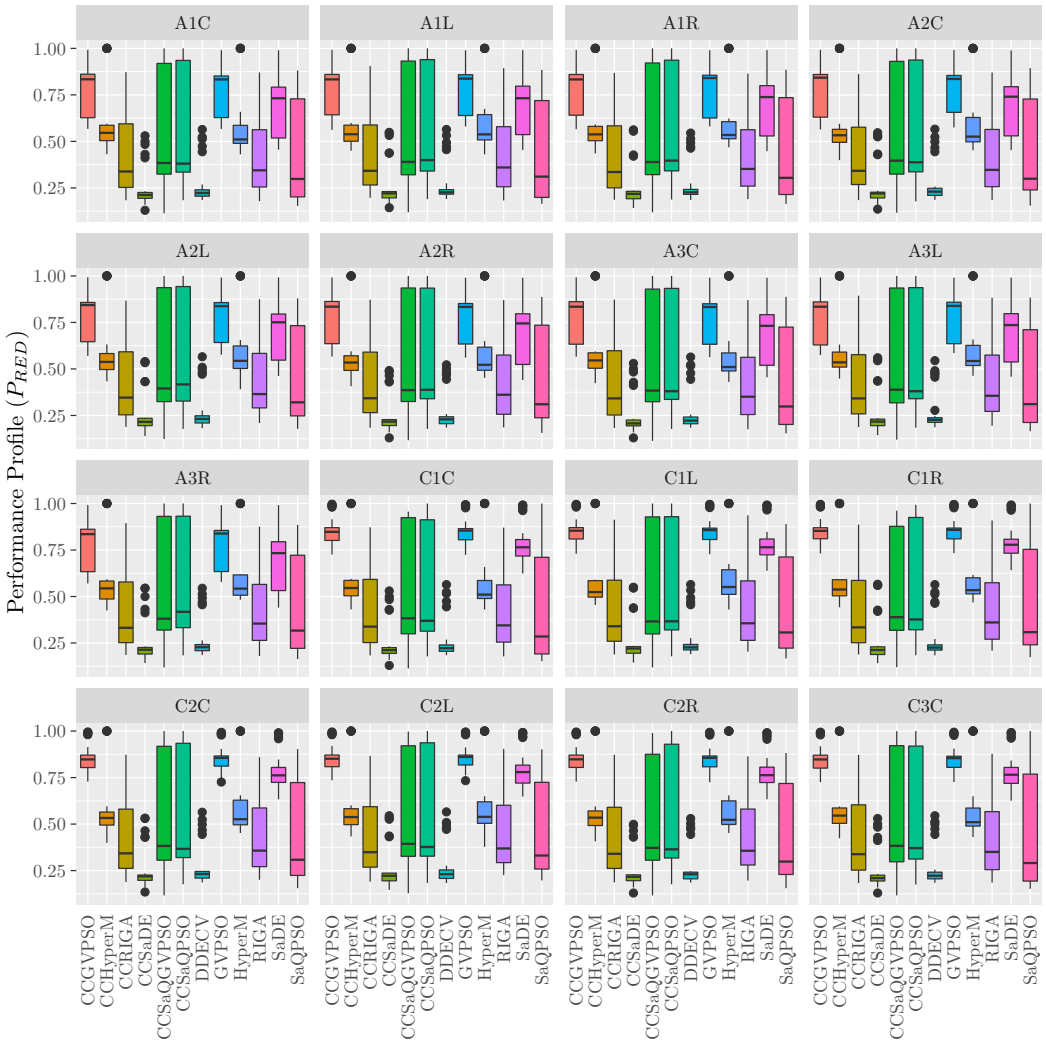


Figure 10.1: Objective space perspective of constraint behaviour spaces: `A1C` to `C3C`

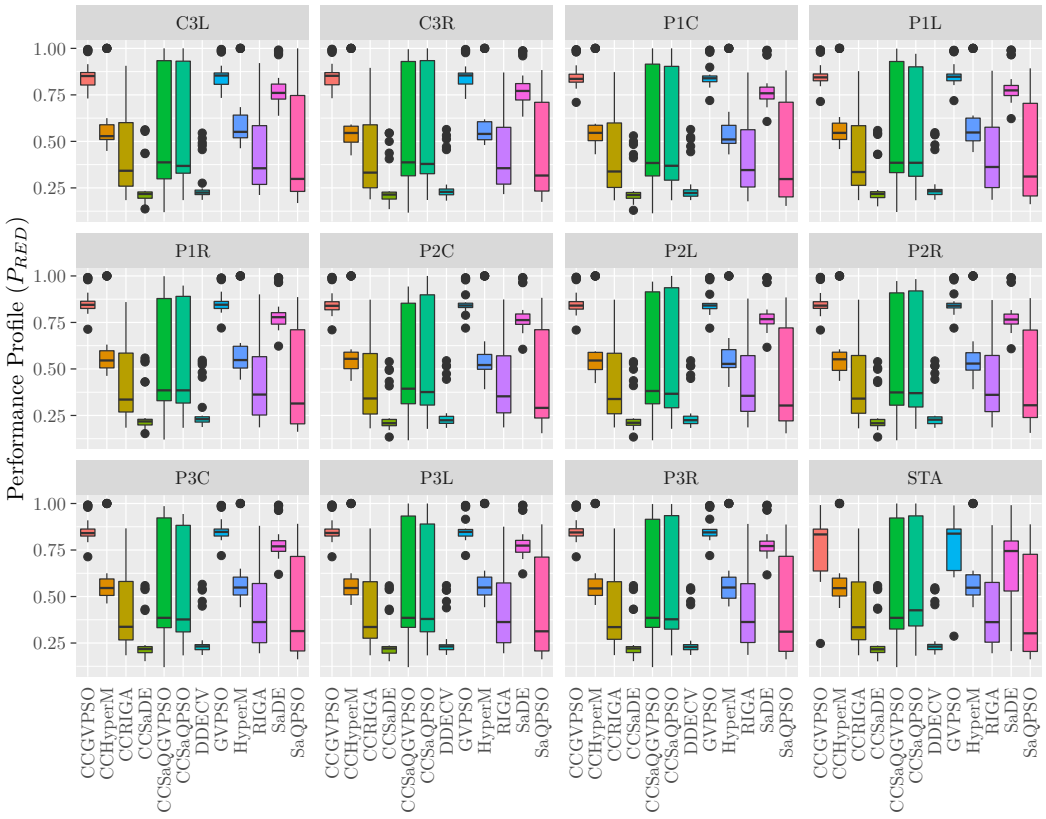


Figure 10.2: Objective space perspective of constraint behaviour spaces: `C3L` to `STA`

10.1.2 Constraint Space Behaviour

The purpose of this section is to analyse the ability to cope with changes in the constraint behaviour space whilst the objective behaviour space is fixed. Similar to the objective space behaviour results, CCSaDE and DDECV produced the best results for the constraint space perspective, illustrated in the box-plots in figure 10.3. From the results, CCSaDE and DDECV are able to provide good results which agree with the observations with respect to the objective behaviour space perspective. The remainder of the algorithms consistently provided results that were less desirable and which varied across the range of P_{RED}. The constraint behaviour space performance of the CCGVPSO, GVPSO and SaDE are consistently part of the worst performing algorithms across all constraint behaviour spaces. More consistent results between the algorithms were present within the progressive objective behaviour spaces.

As mentioned in section 10.1.1, CCGVPSO, GVPSO and SaDE displayed larger inter-quartile ranges for the abrupt objective behaviour spaces as well as the `STA` behaviour space. The only exception to this observation is for the `A1C` constraint behaviour space, which also provided the worst overall P_{RED} results for all the abrupt constraint behaviour spaces. For the progressive objective behaviour spaces (*i.e.*, the `P**` problem instances), SaQPSO presented the largest inter-quartile range. A large difference in performance can be seen between CCSaQPSO and SaQPSO for the progressive constraint behaviour space box-plots. This difference in performance agrees with the degraded performance of SaQPSO observed within the progressive objective behaviour spaces.

Results for constraint behaviour spaces `*1C` and `*3C` indicate that these constraint space behaviours were the most challenging for the optimisation algorithms, with larger P_{RED} measurements. As a result, the algorithms display worse performances for constraint behaviours where the position of optima within the optimisation problem change, whilst displaying a circular movement pattern.

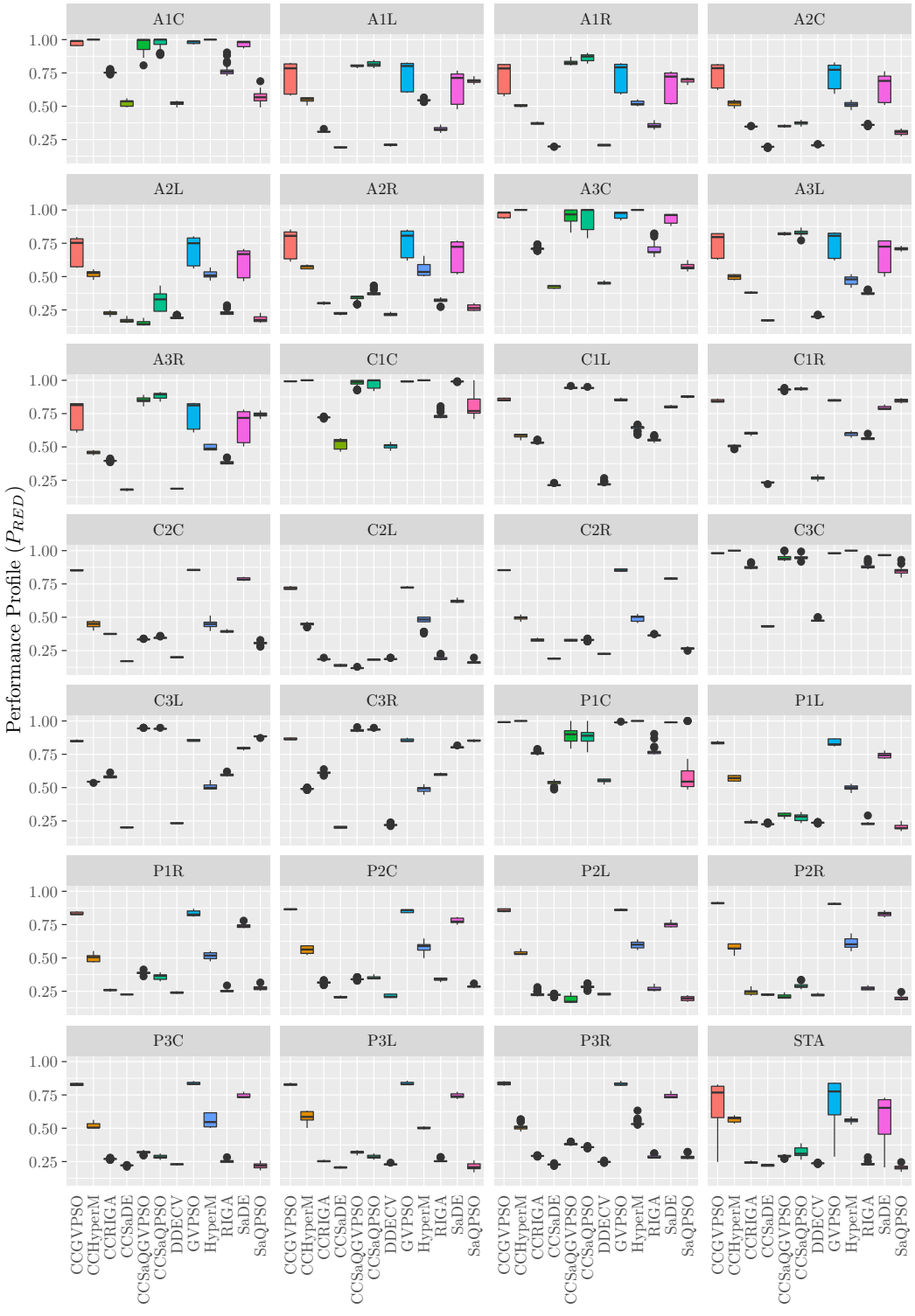


Figure 10.3: Constraint space perspective of objective behaviour spaces

10.1.3 Algorithm Performance on Benchmark Problem Instances

This section discusses the optimisation algorithm accuracy results based on the different main optimisation problem categorisations, related to the characteristics of the individual benchmark problem instances.

Section 10.1.3.1 groups the benchmark problem instances based on the category of DCOP problems. The spatial and temporal characteristics of the benchmark problem instances are used to evaluate optimisation algorithm accuracy in section 10.1.3.2. Lastly, the overall accuracy results for the optimisation algorithms across all of the optimisation problem instances are discussed in section 10.1.3.3.

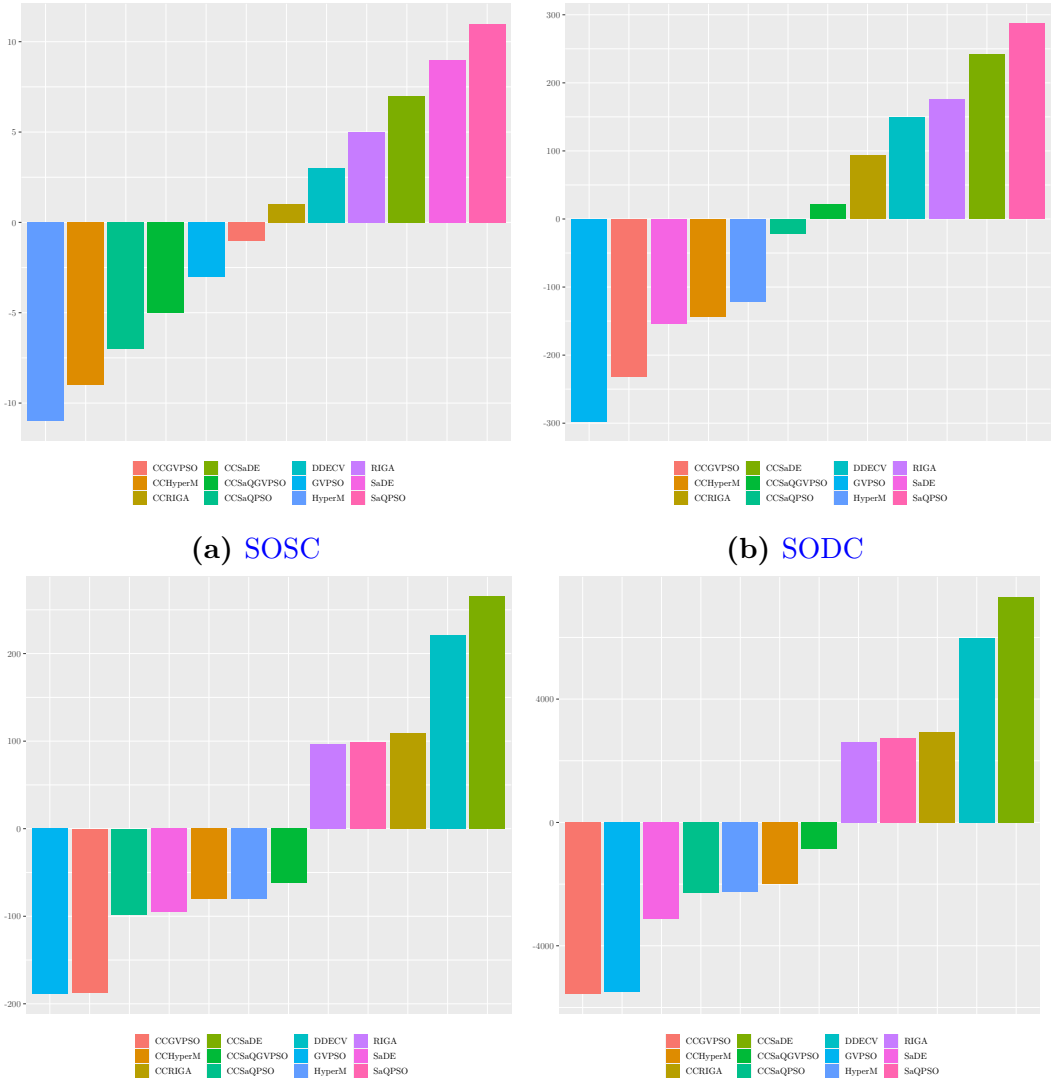
10.1.3.1 Accuracy Based on DCOP Category

To investigate the accuracy of the optimisation algorithms, each category of DCOP is considered individually. The wins and losses for the optimisation algorithms were assigned based on the statistical comparison process described in section 9.5. Figures 10.4a to 10.4d respectively illustrate the win-minus-loss results for the SOSC, SODC, DOSC and DODC categories of the 784 DCOP unique instances.

For the SOSC problem instances, the optimisation algorithms with the largest number of wins are SaQPSO, SaDE, CCSaDE, RIGA, DDECV and CCSaDE. From these "winning" algorithms, the SaQPSO achieved the most wins. Three dynamic co-evolutionary algorithms produced more losses than wins for the SOSC problem instance, namely CCHyperM, CCSaQPSO and CCSaQGVPSO. Considering that the SOSC category of DCOPs is the least complex, it is also the least represented from the 784 unique problem instances. Notably, the SOSC problem instances are static, constrained optimisation problems and do not require any adaptation from the optimisation algorithm nor the constraint handling method.

The SODC category of DCOPs, demonstrate win-minus-loss results that differ greatly from the SOSC results: GVPSO achieved the most losses. The wining optimisation algorithms in descending order are SaQPSO, CCSaDE, RIGA, DDECV, CCRIGA and CCSaQGVPSO. Although the winning optimisation algorithms contain a mixture of both constraint handling methods, the optimisation algorithms that use the α-constraint method achieved a greater number of overall wins based on the combined win-minus-loss results of the optimisation algorithms.

Within the DOSC category of problem instances, the order of algorithms differs from the ordering obtained within the SOSC and SODC categories of DCOPs. The win-minus-loss results demonstrate that the CCSaDE dominated the other algorithms, followed by DDECV as the next best performing optimisation algorithm. Figure 10.4c shows that the win-minus-loss results for the



(a) SOSC (b) SODC

(c) DOSC (d) DODC

Figure 10.4: Algorithm wins and losses for the categories of DCOP instances based on median P_{RED} results

optimisation algorithms that are not on the far left or right of the plot present similar performances, with the GVPSO and CCGVPSO almost achieving the same final loss values. This result is similar to the results presented for the objective space behaviour of CCSaDE and DDECV. However, it should be noted that the objective space behaviour plots consider all 784 problem instances instead of just the DOSC group of problem instances.

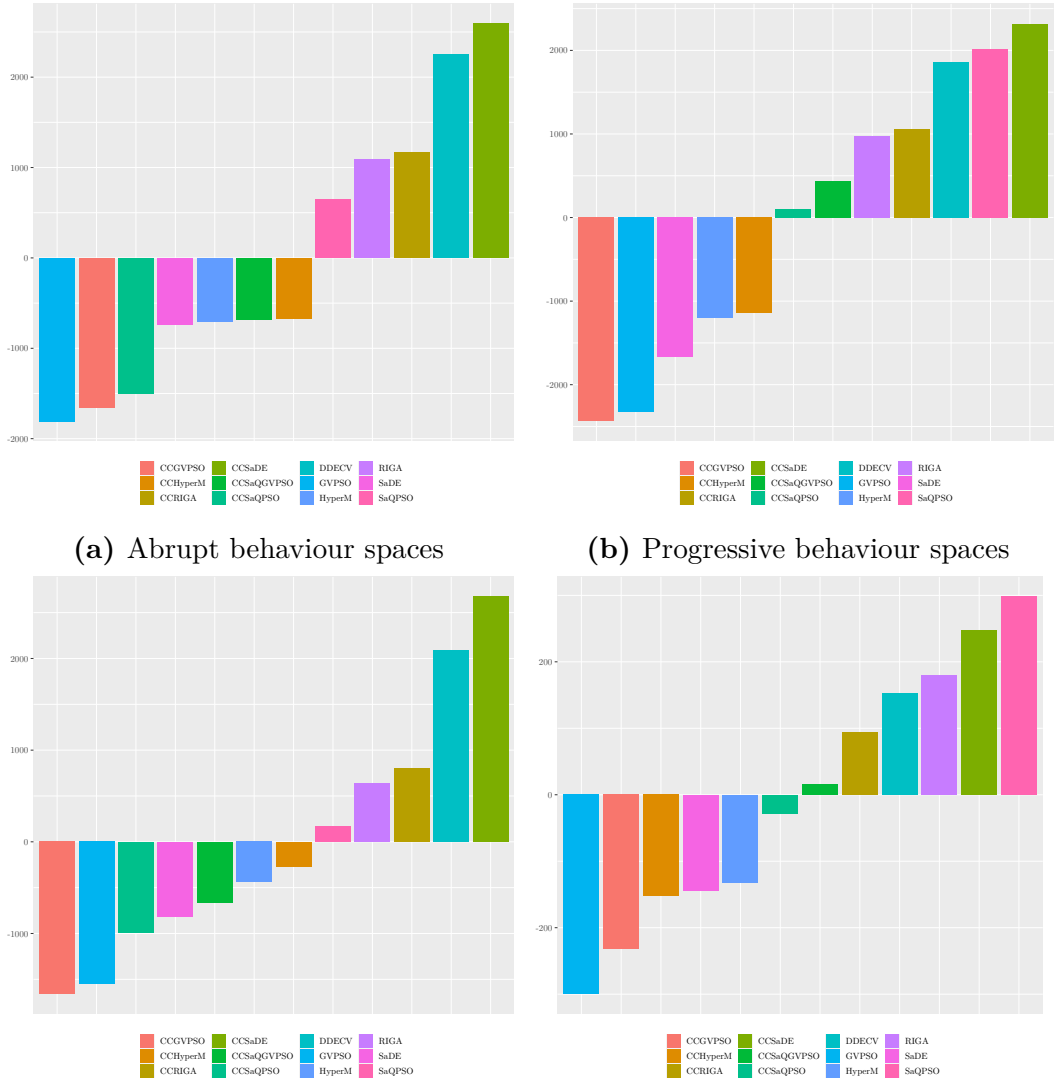
For the most complex category of problem instances, namely the DODC instances, the algorithm ordering based on the wins-minus-losses nearly matches that of the DOSC problem instances. The winning optimisation algorithms present the same order as the DOSC optimisation algorithms, with the actual win-minus-loss values being larger because the majority of the DCOP instances fall into this category. The optimisation algorithm order for the algorithms achieving a larger number of losses is different to the optimisation algorithms within the DOSC problem instances. The SaDE achieved more losses than wins to provide the third worse algorithm performance, whilst CCSaQGVPSO achieved the least number of losses. These results indicate that the winning optimisation algorithms provided better performances and achieved more wins for more of the problem instances.

10.1.3.2 Accuracy for Temporal and Spatial Severity Problem Categories

Figures 10.5 and 10.6 respectively illustrate the algorithm performances based on the severity of spatial and temporal changes of the objective and constraint behaviour spaces. The benchmark problems where categorised based on the categories defined by Duhain and Engelbrecht [77].

For the objective behaviour spaces, figures 10.5a to 10.5d present the optimisation algorithm wins-minus-losses for each spatial and temporal change category. For all change types that have a temporal change severity, the results indicate that CCSaDE and DDECV remain the best performing optimisation algorithms, particularly for abrupt and chaotically changing problem instances. Across all spatial and temporal severity categories for the objective space behaviour plots, CCGVPSO and GVPSO achieved the worst win-minus-loss results. However, for the static and progressive problem instances, SaQPSO displays better win-minus-loss results by achieving the second best performance for the progressive problem instances whilst performing the best for the static objective space behaviour. Because the problem landscape changes within progressive problem instances are small but frequent, SaQPSO using the α-constraint method indicates that it is able to successfully track the current solutions.

The constraint behaviour spaces in figure 10.6, regardless of the spatial and temporal change category, indicate that the best performing optimisation algorithms are CCSaDE and DDECV. Optimisation algorithm ordering for the win-minus-loss results remained largely the same, but the abrupt and chaotic



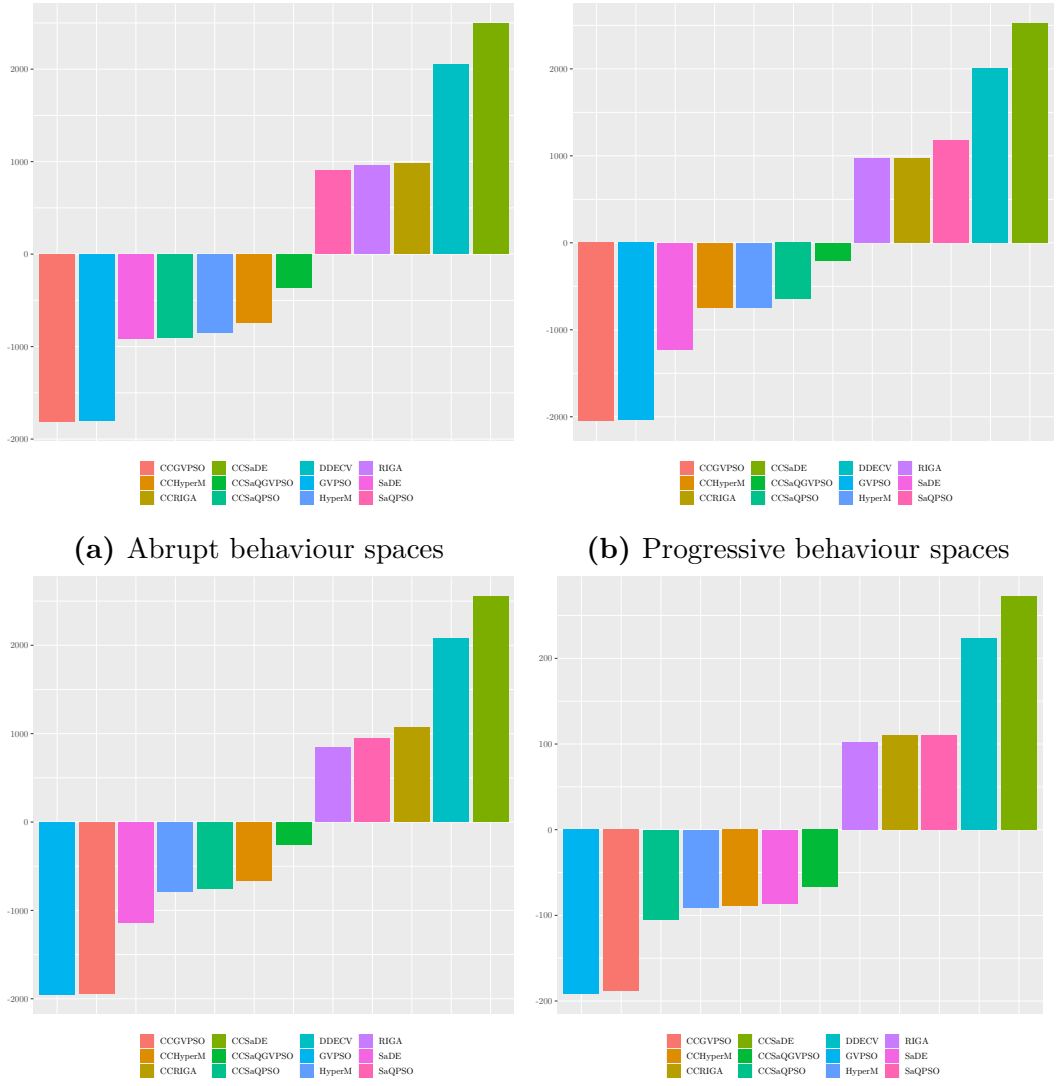
(a) Abrupt behaviour spaces

(b) Progressive behaviour spaces

(c) Chaotic behaviour spaces

(d) Static behaviour spaces

Figure 10.5: Algorithm preference based on objective behaviour space DCOP classes



(a) Abrupt behaviour spaces

(b) Progressive behaviour spaces

(c) Chaotic behaviour spaces

(d) Static behaviour spaces

Figure 10.6: Algorithm preference based on constraint behaviour space DCOP classes

problem instances show that RIGA obtains slightly better than SaQPSO for abruptly changing spaces whereas the SaQPSO is preferred for chaotic problem instances. In contrast, for the progressive and static problem instances, the only changes in optimisation algorithm ordering occur within the algorithms that achieved more losses. The winning algorithms, excluding CCSaDE and DDECV, within the progressive and static problem instances display similar win-minus-loss results for both categories with SaQPSO performing better within the progressive problem instances.

10.1.3.3 Accuracy Across All Benchmark Problem Instances

The overall win-minus-loss results across all benchmark problem instances are illustrated in figure 10.7, which indicates that the best performing algorithm is CCSaDE, followed by DDECV, SaQPSO, CCRIGA and RIGA. From these best performing algorithms, DDECV, SaQPSO and RIGA used the α-constraint method. The optimisation algorithms that obtained more losses were CCGVPSO and GVPSO. For the remainder of the algorithms with more losses than wins, the least number of losses are obtained by the dynamic co-evolutionary algorithms.

Performance results for all algorithms across the 784 instances is provided in table A.1 which contains the win-minus-loss information. Additionally, the number of changes experienced by each objective space behaviour and constraint space behaviour combination is also provided. From the individual instance data, the progressive objective behaviour spaces tend to favour SaQPSO and is confirmed in figure 10.5b. No discernible pattern can be observed to indicate which combination of objective space behaviour and constraint space behaviour would be favoured by DDECV. The largest number of wins for DDECV present themselves within the `A2R` objective behaviour space, with additional winning performances achieved for the `C1C` and P2R objective behaviour spaces. For the remainder of the benchmark problems, CCSaDE displays dominant performance results. A smaller number of problem instances are won by either RIGA or CCRIGA, with CCRIGA winning slightly more instances between these algorithms.

Figure 10.8 illustrates the overall statistical performance of the algorithms across all DCOP benchmark problem instances. It is clear that CCSaDE provides a statistically significant performance by being the only optimisation algorithm to be contained within the critical difference band. With the exception of DDECV and CCSaQPSO, the remaining algorithms can be grouped together in clusters which indicate equivalent performances. The best performing cluster contains the SaQPSO, CCRIGA and RIGA algorithms with the remaining four clusters providing results that are poorer than those obtained by CCSaQGVPSO. The ordering of algorithms from best to worst (from left to right on the critical difference plot) shows that more dynamic co-evolutionary algorithms are on the left hand side when compared to the other algorithms.

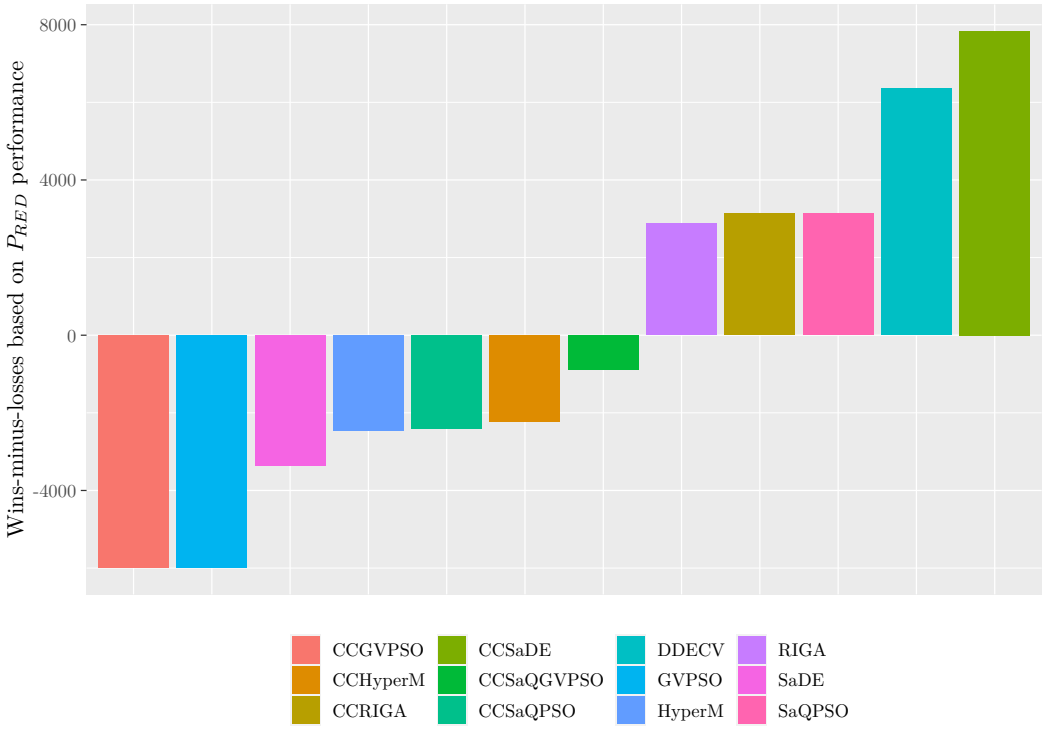


Figure 10.7: Overall algorithm wins and losses across all DCOP instances based on median P_{RED} results

Therefore, except for the performances of DDECV, SaQPSO and RIGA, the dynamic co-evolutionary algorithms provide preferred algorithm performances across the 784 DCOP problem instances. Moreover, the only statistically significant result was obtained by an algorithm that makes use of the Lagrangian reformulation of the optimisation problem, namely CCSaDE.

10.1.4 Sampled Individual Problem Performances

The previous section discussed the performance of the proposed dynamic co-evolutionary algorithms, and dynamic optimisation algorithms with α-constraint, across all the optimisation problem benchmark instances. Because the DODC category of DCOP instances represents the most difficult optimisation problems, the results within this section focus on DODC problems only.

In this section, an analysis is performed for specific problem instances within the DODC category of instances. This is done to analyse the performance for easy to more difficult within the DODC category of problem instances. The problem instances were selected based solely on the characteristics of the objective space behaviour and constraint space behaviour. The selected sample of problem instances, in increasing complexity, are

- `A1L/A1L` (Low)

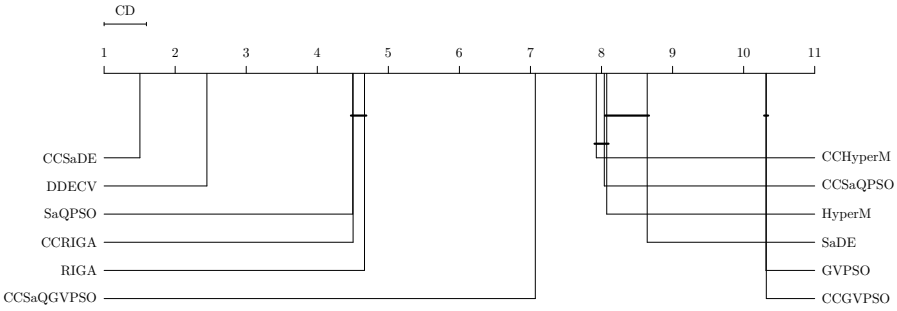


Figure 10.8: Critical different plot of overall algorithm performance across all problems

- `P2C/P2C` (Medium)
- `C3R/C3R` (High)

Figure 10.9 illustrates the algorithm win-minus-loss performances for each of the above mentioned problem benchmark instances. Across the these problem instances, CCSaDE and DDECV produced the most number of wins, mirroring the general trend observed across all problem instances in section 10.1.3.3. For the other algorithms, the results indicate varied performances for each of the individual benchmark problem instances. The `A1L/A1L` problem instance demonstrates that SaDE is able to achieve wins, albeit rare when based on the performances within appendix A. The SaQPSO provides a respectable performance for the `P2C/P2C`, whilst also showing that the variants CCSaQGVPSO and CCSaQPSO can also provided better performances for the progressive problem instance. For `C3R/C3R`, both HyperM and CCHyperM performed well even if the overall performance results indicate contrary results across all problem instances. For each of the sampled individual problem instances, CCSaDE was the best performing algorithm with the most wins, followed by DDECV.

The algorithm performance profiles, based on the P_{RE} measure, for the sampled individual problem instances are illustrated in figure 10.10. For the `A1L/A1L` problem instance, CCSaDE and DDECV start the optimisation process by improving on the relative error of the best candidate solution. Around algorithm iteration 150 both CCSaDE and DDECV begin to settle on their performances and maintain this result, with minor observed fluctuations. The general trend of improvement is seen for all algorithms except for CCSaQPSO and SaQPSO, which both initially show improved results before deteriorating to achieve the worst final results. For the `P2C/P2C` instance, CCSaDE and DDECV

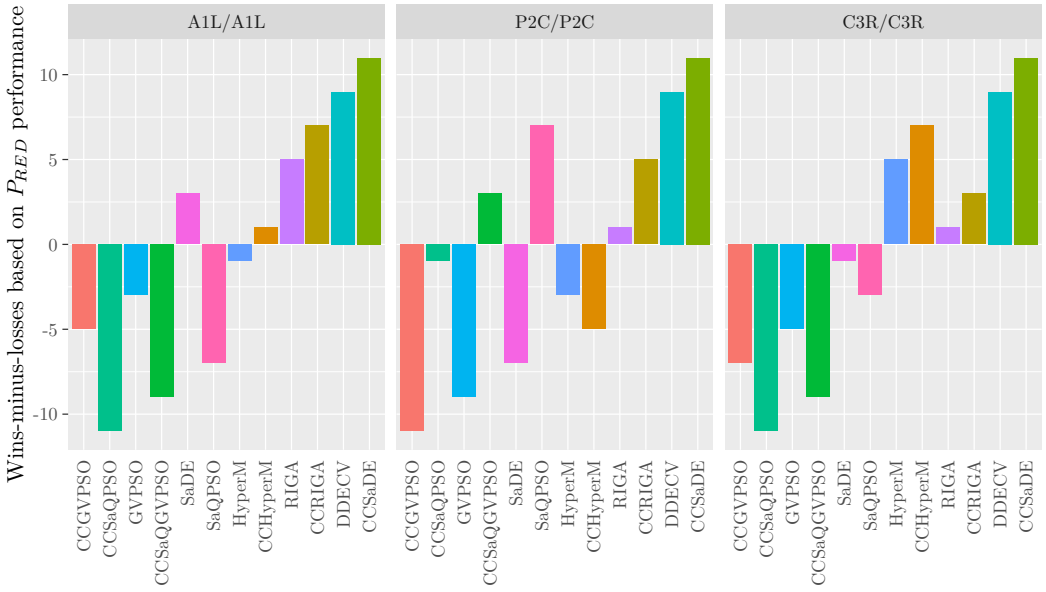


Figure 10.9: Algorithm performances for individual low, medium and high complexity benchmark problem instances

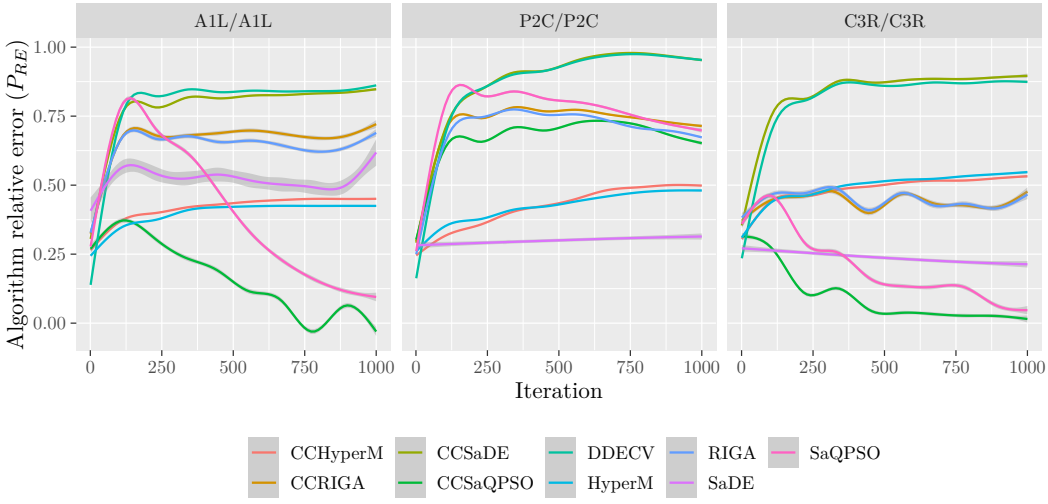


Figure 10.10: Algorithm relative error per iteration for individual CMPB benchmark problems

once again show improvement in the observed results, with the improvement continuing across the algorithm iterations. In addition, the CCHyperM, HyperM and SaDE show improvements. However, the SaDE improvement is marginal. The remaining algorithms display an initial improvement, but degrade from the initial success over the course of algorithm execution. For the `C3R/C3R` problem instance a noticeable performance difference is observed, with only CCSaDE and DDECV able to produce good quality solutions. Poor

solutions are obtained from CCSaQPSO and SaQPSO for the `A1L/A1L` and `C3R/C3R` benchmark problem instances. Overall, the performance profile results for the sampled individual problem instances agree with the win-minus-loss results in section 10.1.3.3.

10.1.5 Conclusion

The performance results indicate that accurate solutions are produced by all optimisation algorithms, but only CCSaDE and DDECV produced solutions that are close to the desired target solution based on the P_{RED} measurement. Objective behaviour spaces and constraint behaviour spaces have a significant impact on algorithm performance, as shown in sections 10.1.3.1 and 10.1.3.2 where the categories of DCOPs were isolated. When considering the objective behaviour space and constraint behaviour space, the objective behaviours displayed more variability with respect to algorithm preference when compared to the constraint behaviour spaces. Although CCSaDE achieved the overall best algorithm performance for the benchmark problems, SaQPSO displayed improved performances for the progressive and the static objective behaviour spaces.

Examination of the individual problem instances highlighted the performance of the algorithms across iterations. The abrupt and chaotic objective behaviour spaces indicated a preference for CCSaDE and DDECV, whilst the static and progressive objective behaviour spaces also included SaQPSO as an algorithm that performs well. Because progressive landscapes maintain a sequence of small changes to the optimisation problem, SaQPSO was able to provide solutions to these problem landscapes more efficiently. In contrast, for abrupt and chaotic objective behaviour spaces, where the frequency of change is larger, the win-minus-loss results of SaQPSO demonstrated that the algorithm achieved more losses. Because CCSaDE and DDECV are able to produce new candidate solutions in areas that are not restricted by the quantum cloud, the amount of exploration in these algorithms is larger than that for SaQPSO.

The constraint handling methods within the algorithms do influence the final results (from the objective behaviour space perspective), but the impact of the constraint handling is overshadowed by the ability of the algorithm to adapt to problem landscape changes. Without adequate adaption, the algorithms are unable to produce better performances even when the constraint handling method encourages feasible solutions. Both CCSaDE and DDECV indicate that they are able to adapt to problem landscape changes and are thus able to produce better performances. The better performance is in spite of the fact that different constraint handling methods are used in the CCSaDE and DDECV algorithms.

10.2 Analysis of Algorithm Diversity

The diversity of candidate solutions provides an indication of the degree of homogeneity, with low homogeneity indicating a state of exploration, and high homogeneity indicating a state of exploitation. Tracking the diversity is especially helpful for dynamic optimisation algorithms, where larger diversity levels aid the optimisation algorithm in coping with change in the optimisation problem. This section aims to determine how the diversity of candidate solutions influences the performance of the optimisation algorithms. As with the accuracy result analysis in section 10.1, the diversity of the optimisation algorithms is considered by taking both objective behaviour space and constraint behaviour space into consideration.

This section presents the diversity of the objective behaviour space in section 10.2.1, followed by the diversity of the constraint behaviour space in section 10.2.2. An analysis of the diversity for individually sampled problem instances, introduced in section 10.1.4, is given in section 10.2.3. This section concludes with section 10.2.4.

10.2.1 Objective Behaviour Space

The purpose of this section is to analyse the levels of diversity with objective space changes whilst the constraint behaviour space is fixed. The diversity of the optimisation algorithms across all benchmark problems, grouped by the objective behaviour space, is illustrated in figure 10.11.

Across the objective behaviour spaces, the majority of the optimisation algorithms display a fairly stable amount of diversity throughout the entire execution of the optimisation algorithms. As a result, the optimisation algorithms continue to explore the optimisation problem search space because the homogeneity of the candidate solutions does not drop below this stable diversity level. The majority of the algorithms had larger starting diversity values which reduced over time to an oscillating diversity value, before algorithm stopped execution at the last iteration. Both HyperM and SaDE maintained an almost constant amount of diversity for all objective behaviour spaces.

The largest amount of diversity was present within the SaQPSO for most of the object behaviour spaces. However, for the progressive objective behaviour spaces the amount of diversity within SaQPSO continued to increase, indicating diverging particle behaviour within the algorithm. Due to the diverging particles, SaQPSO was not able to provide a solution and the exploding diversity values for SaQPSO may indicate a failure in the constraint handling method within the algorithm. On the other hand, this observation makes no distinction between the neutral and quantum particles of SaQPSO. The size of the quantum cloud (*i.e.*, r_{cloud}) self-adapts throughout the execution of the algorithm, and for the progressive environments the size of the quantum cloud continued to increase. Therefore, the amount of diversity within the SaQPSO

increases without limit even though SaQPSO was able to obtain solutions to the progressive problem instances. For the non-progressive problem instances the SaQPSO was able to initially increase the diversity and then reduce the diversity as the algorithm execution approached termination. These results indicate that although SaQPSO was able to provide greater diversity levels than the other algorithms, it was not always possible to prevent the diversity from exploding to large values.

The continued divergence observed with SaQPSO occurs due to the self-adaption of the r_{cloud} control parameter. By taking the maximum between the neural particle diversity and the quantum particle diversity (see equation (7.3)), a divergent behaviour can occur when particles roam and do not have enough time to return to the defined problem bounds. In this case, the diversity of the neutral particles can be larger than that of the quantum particles. Thereafter, the quantum cloud centred at the current best particle can attract particles to positions further away from the defined problem domain. This behaviour is exacerbated further by the loss of information after a change in the problem landscape, when neutral particle memory is re-evaluated. The abruptly changing behaviours observe lower diversity with fewer changes in the optimisation problem and allows more time for roaming particles to return.

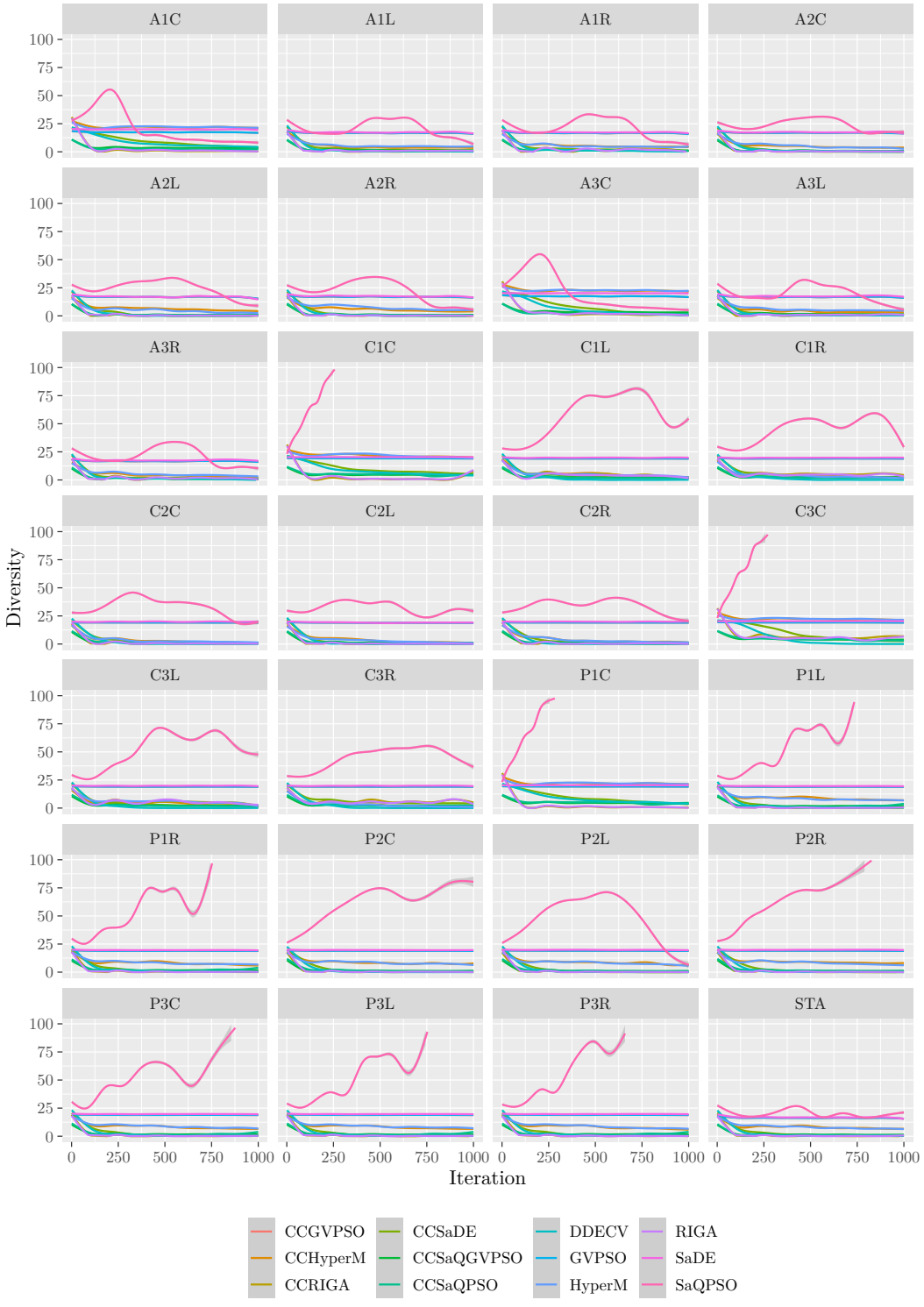


Figure 10.11: Algorithm diversity across objective behaviour spaces

10.2.2 Constraint Behaviour Space

This section analyses the levels of diversity with constraint space changes whilst the objective behaviour space is fixed. The diversity for the constraint behaviour spaces is provided in figure 10.12.

The observed diversity within these box-plots indicate that more reasonable diversity values are obtained for the optimisation algorithms. That is to say that the observed diversity of the optimisation algorithms appears to be similar to that of the objective behaviour space perspective (see section 10.2.1).

For all the optimisation algorithms, except for SaQPSO, the observed diversity levels remain flat for the majority of the algorithm iterations. The observed diversity of these algorithms are similar diversity profiles than that of the objective behaviour space with SaDE and HyperM having larger continuous diversity values.

For SaQPSO, the constraint behaviour spaces of `P1L` and `P1R` display ever increasing diversity values. As discussed in section 10.2.1, the divergent behaviour is due to the combined effect of frequent optimisation problem changes, resulting in repeated particle roaming, and the use of the maximum diversity between neutral and quantum particles. The constraint behaviours of `C1C` and `P3R` appear to begin to diverge, however a reduction in the diversity is observed just before the final algorithm iteration. The observed diversity for the abruptly changing problem search spaces as well as for the static constraint behaviour space demonstrate lower general diversity levels for the SaQPSO. Fewer optimisation problem changes is observed for the abruptly changing problem search spaces and for the static behaviour space, allowing for the return of roaming particles.

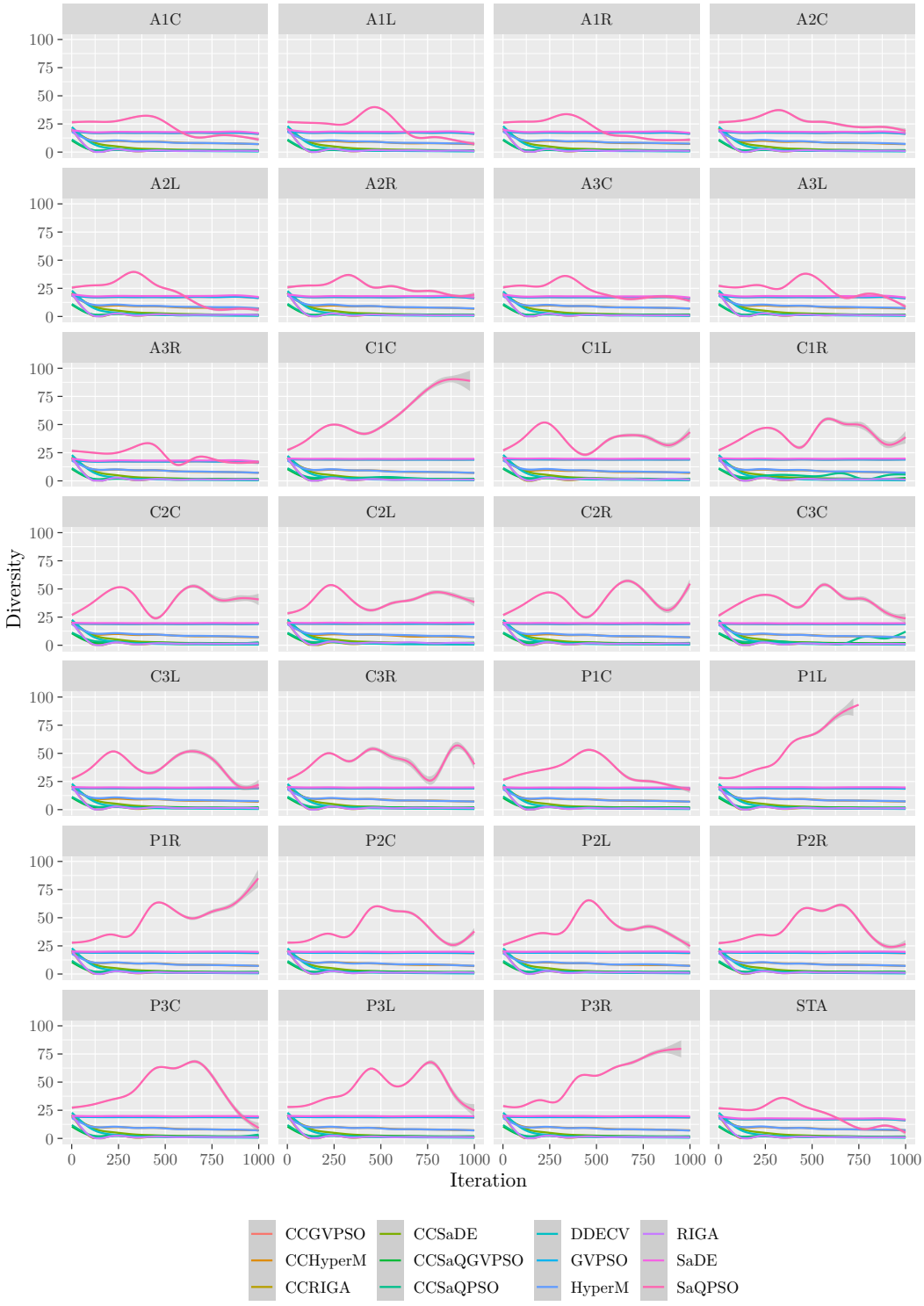


Figure 10.12: Algorithm diversity across constraint behaviour spaces

10.2.3 Diversity of Individual Problem Instances

Figure 10.13 illustrates the diversity profiles for the optimisation algorithms on the individually selected problem instances. Across all three problem instances, the diversity of SaDE and HyperM remained larger than almost all other algorithms without any indicated reduction in diversity. The diversity of SaQPSO was initially the largest diversity value, but does not remain at that level for many iterations. For `A1L/A1L`, SaQPSO reduced diversity over time to an almost constant value that is slightly larger than the majority of the tested algorithms. However, for `P2C/P2C` and `C3R/C3R`, the diversity of SaQPSO continued to increase as demonstrated in sections 10.2.1 and 10.2.2. Diversity for best performing optimisation algorithms, CCSaDE and DDECV, continued to oscillate across algorithm iterations with figure 10.13 not being granular enough to make the oscillation visible.

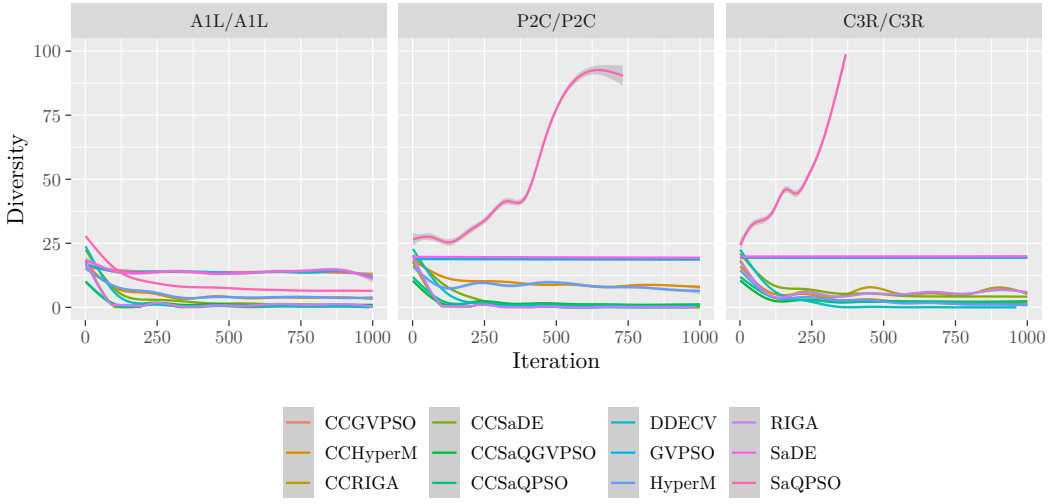


Figure 10.13: Diversity of individual problem instances

10.2.4 Conclusion

For all optimisation algorithms different levels of diversity can be observed across all iterations. With the better performing optimisation algorithms, the diversity values do not approach zero whereas the optimisation algorithms that achieved worse accuracy performances display a reduction in diversity. As the diversity of the candidate solutions decreases, the homogeneity of the candidate solutions increases. Therefore, a consistent amount of diversity provides improved exploration of the optimisation problem search space and allows for the discovery of new solutions.

The algorithm with the most difficulty in managing and maintaining diversity was SaQPSO, which had ever increasing diversity values displayed within

the objective behaviour spaces. Possible explanations have been provided for this behaviour, but does motivate further investigations to understand this behaviour.

10.3 Algorithm Recovery in Dynamic Constrained Optimisation Problems

Changes within the optimisation problem are disruptive to the optimisation process, but especially to the optimisation algorithm. For optimisation algorithms to cope with the changes in the optimisation problem landscape, the candidate solutions show be adjusted to reflect the updated problem search space to allow for the continued search for better candidate solutions. Multiple performance measures have been suggested as indicators of the resilience of an optimisation algorithm to changes within the optimisation problem. This section considers three performance measures to understand the effect that changes in optimisation problem have on the optimisation algorithm with reference to DCOPs.

The ability of the optimisation algorithms to recover before a change is experienced by the optimisation problem is discussed in section 10.3.1, whilst the recovery after experiencing a change in the optimisation problem is discussed in section 10.3.2. Section 10.3.3 discusses whether optimisation algorithms are able to recover across the individual changes experienced in the optimisation problem. The overall recovery performing is discussed in section 10.3.4. Optimisation algorithm recovery for the three individually sampled problem instances is provided in section 10.3.5. Lastly, section 10.3.6 concludes this section.

10.3.1 Algorithm Resilience Before Landscape Changes

The observed errors preceding a landscape change are investigated with the P_{ABEBC} performance measure (see section 6.2.2.2). Figure 10.14 provides boxplots of algorithm performance, based on the objective space behaviour, just before the optimisation problem experiences a change. From the accuracy results in section 10.1, CCSaDE and DDECV provide the best performances for most of the object space behaviours based on the P_{RE} performance measure. The inter-quartile range for CCSaDE and DDECV is shorter than that of the other algorithms for the majority of the objective space behaviours. The shorter inter-quartile range implies that a smaller amount of variance exists within the results of CCSaDE and DDECV. Larger variances in the performance of CCSaDE and DDECV is observed within the `*1C` objective space behaviours, suggesting greater challenges for the optimisation algorithms to continue to track and locate solutions. The CCHyperM and HyperM algorithms displayed

the largest inter-quartile ranges for all objective space behaviours and demonstrated performances that vary from not finding a solution at all to finding solutions, albeit that the solutions are sub-optimal. As with the observed accuracy performances in section 10.1 and diversity in section 10.2, SaQPSO achieved better results within the progressive object space behaviours, with only `P1C` displaying large variance results.

For the constraint space behaviours, the box-plots demonstrate similar performances for the optimisation algorithms and is illustrated in figure 10.15. Although CCSaDE and DDECV have short inter-quartile ranges, they are the only algorithms that displayed the most number of outlier results. The result for the constraint space behaviour does not disagree with the optimisation algorithm performance results that are observed in previous sections.

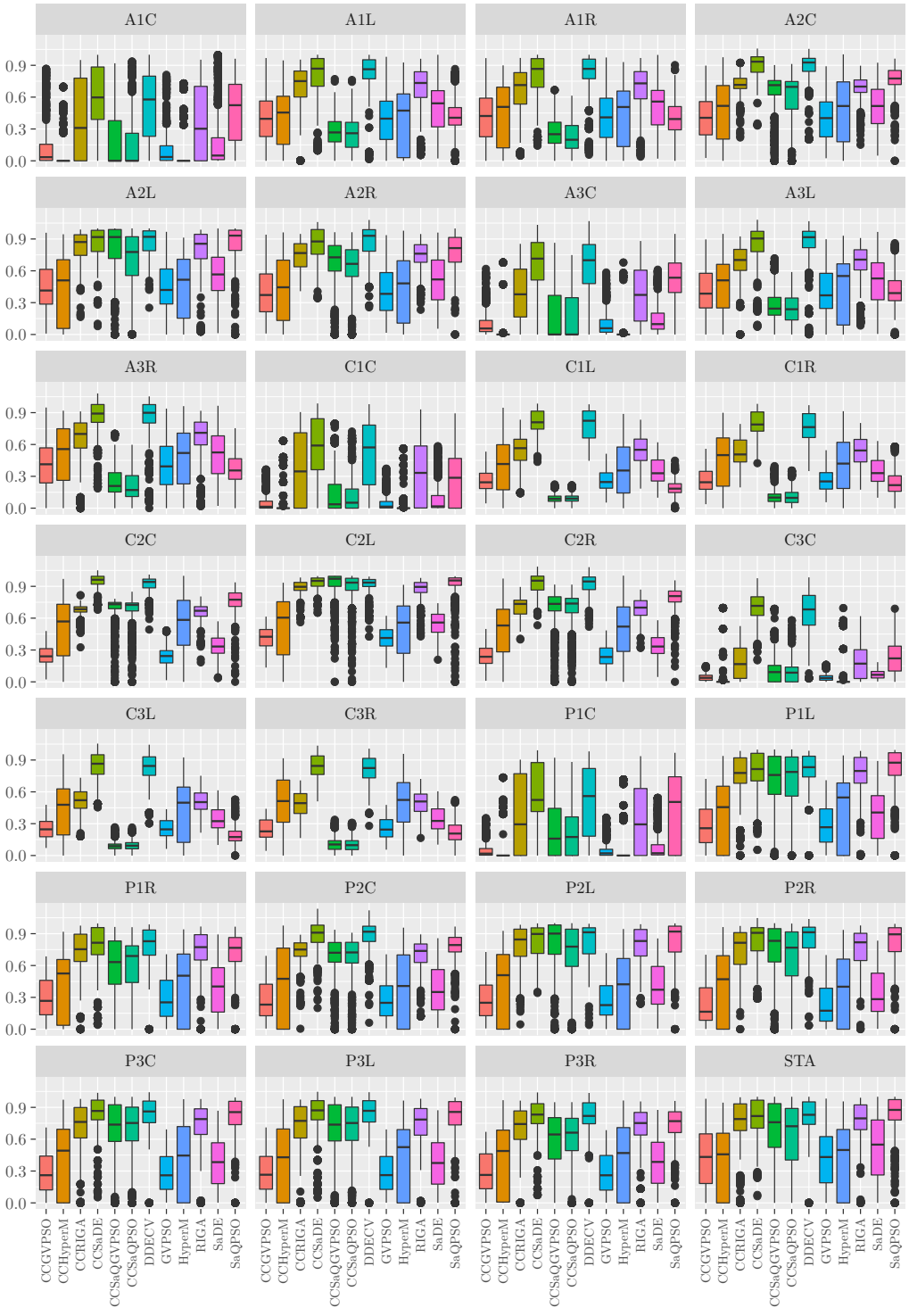


Figure 10.14: ABEBC for algorithms across benchmark problem instances from the objective landscape perspective

Figure 10.15: ABEBC for algorithms across benchmark problem instances from the constraint landscape perspective

10.3.2 Algorithm Resilience After Landscape Changes

Once the optimisation problem search space has undergone a change, the optimisation algorithm needs to adapt in order to allow the search process to continue. The P_{ABEAC} (see section 6.2.2.3) measures the P_{RE} of the candidate solutions directly after the optimisation problem has experienced a change and indicates how stable the optimisation algorithm is after the optimisation problem experiences change.

For the objective space behaviours, CCSaDE and DDECV demonstrate performance results that are better or comparable to the other algorithms across objective space behaviours. The type II problem instances (*i.e.*, the `*2*` problem instances) demonstrate better P_{RED} results for the majority of the optimisation algorithms. This should be expected, because optima changes for type II problem instances allow for the change in optima value but the position of optima within the optimisation problem remain unchanged. The CCHyperM and HyperM algorithm results indicate that these algorithms produced the most varied performance results after the change to the optimisation problem, suggesting that the problem space changes affect these algorithms the most. Compared to the P_{ABEBC} results, a larger number of outlier results are present with the P_{ABEAC} performance results.

The constraint space behaviour box-plots, given in figure 10.17, mirror the findings of the constraint space behaviours for the P_{ABEBC} measure. Across all constraint space behaviours, the results are similar for all optimisation algorithms where CCSaDE and DDECV achieved the best P_{RED} results, albeit with outlier results. The worst performing optimisation algorithms for the all constraint space behaviours are CCGVPSO, GVPSO and SaDE. These performances indicate that these algorithms are not as stable as the other optimisation algorithms after a change in the optimisation problem.

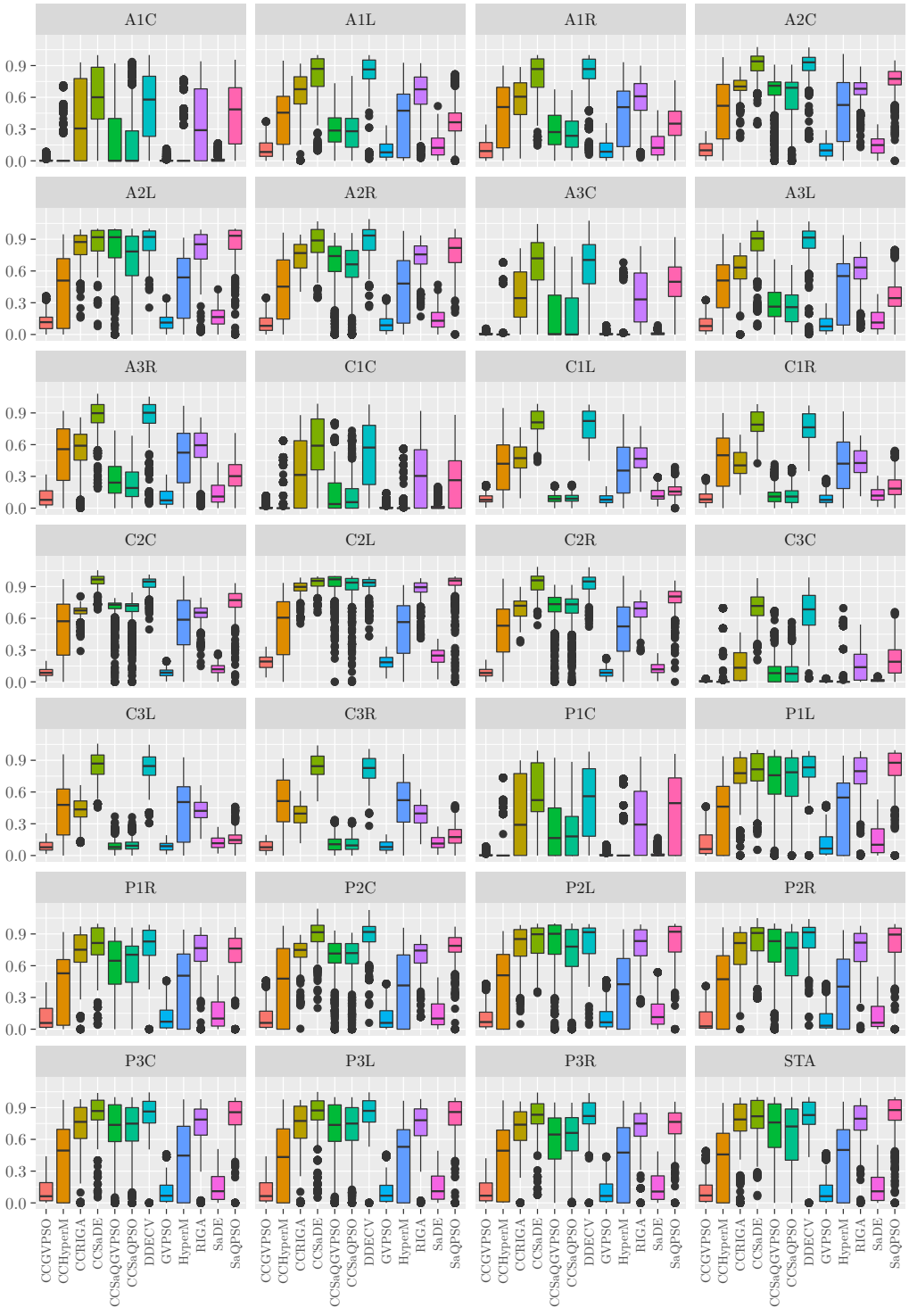


Figure 10.16: ABEAC for algorithms across benchmark problem instances from the objective landscape perspective

Figure 10.17: ABEAC for algorithms across benchmark problem instances from the constraint landscape perspective

10.3.3 Algorithm Recovery Between Landscape Changes

Between subsequent optimisation problem landscape changes the search space remains static. The period of time between subsequent changes to the optimisation problem also defines the *change period* of the optimisation problem and each change period defines a unique static optimisation problem. Within these change periods, an optimisation algorithm may explore and refine solutions before the problem landscape changes, thereby forcing the optimisation algorithm to restart the search process for the changed problem landscape. The P_{ARR} measure (see section 6.2.2.5) quantifies how well the optimisation algorithm is able to recover to the global optimum within a change period. However, with the benchmark problems used within the experimental work, it is not a certainty that an optimum within the objective search space is present within the resultant composed problem search space. Optima can fall into infeasible regions of the final optimisation problem once the constraint search space has been composed.

Consider the top-down projections of the sequence of nine consecutive problem search spaces of the CMPB illustrated in figure 5.1. As a consequence of the stochastic nature of the landscape changes within both the MPB and CMPB no guarantee can be made to ensure that the composed problem search space will always have optima in feasible regions. This uncertainty increases as the number of peaks within the objective problem space for the CMPB approaches zero, or when the number of constraints within the constraint search space increases to cover more of the problem domain. When no optima in the composed problem search space are feasible, the best possible performance that can be obtained is that of the base problem landscape. Within the two-dimensional Cartesian space, the base problem landscape for the MPB generator function is found where $y = 0$. A solution found on the base problem landscape still provides a better solution when compared to infeasible solutions. After all, the found solution is within the feasible region of the composed problem search space of the CMPB.

Figures 10.18 and 10.19 respectively provide P_{ARR} results for the objective space behaviours and the constraint space behaviours, for the optimisation algorithms considered in the experimental work. For the objective space behaviours, the results presented in figure 10.18 indicate that the algorithms were not able to consistently recover to the expected global optimum within a given change period. Better P_{ARR} results are present for the abruptly changing objective space behaviours, with the progressive and chaotic behaviours showing a noticable reduction in P_{ARR} value. Referring to the number of changes experienced by each problem instance in appendix A for the objective behaviour spaces, a minimum of 10 to a maximum of 50 search space changes are experienced for the abruptly changing objective behaviour space. A similar improvement in P_{ARR} value is visible for the `STA` objective behaviour space

which has a maximum number of 33 problem changes. However, the progressive and chaotic objective behaviour spaces report that a minimum of 50 changes occur across all optimisation algorithm iterations. By considering both the frequency of change within the DCOP instances together with the problem instance complexity, the recovery of the optimisation algorithms do indicate some recovery within the problem change period. The recovery is, however, not always to the expected global optimum of the objective search space, which is measured by the P_{ARR} measure. Moreover, shorter change periods prevent the optimisation algorithm from locating improved solutions, before the next change is experienced.

Similar recovery is observed for the optimisation algorithms within the constraint behaviour spaces presented in figure 10.19. The abruptly changing and `STA` constraint space behaviours indicate increased algorithm recovery. Progressive and chaotic constraint behaviour spaces indicate a reduced recovery, but as discussed for the objective space behaviours, the results are misleading when all behaviour characteristics of the problem instances are not considered.

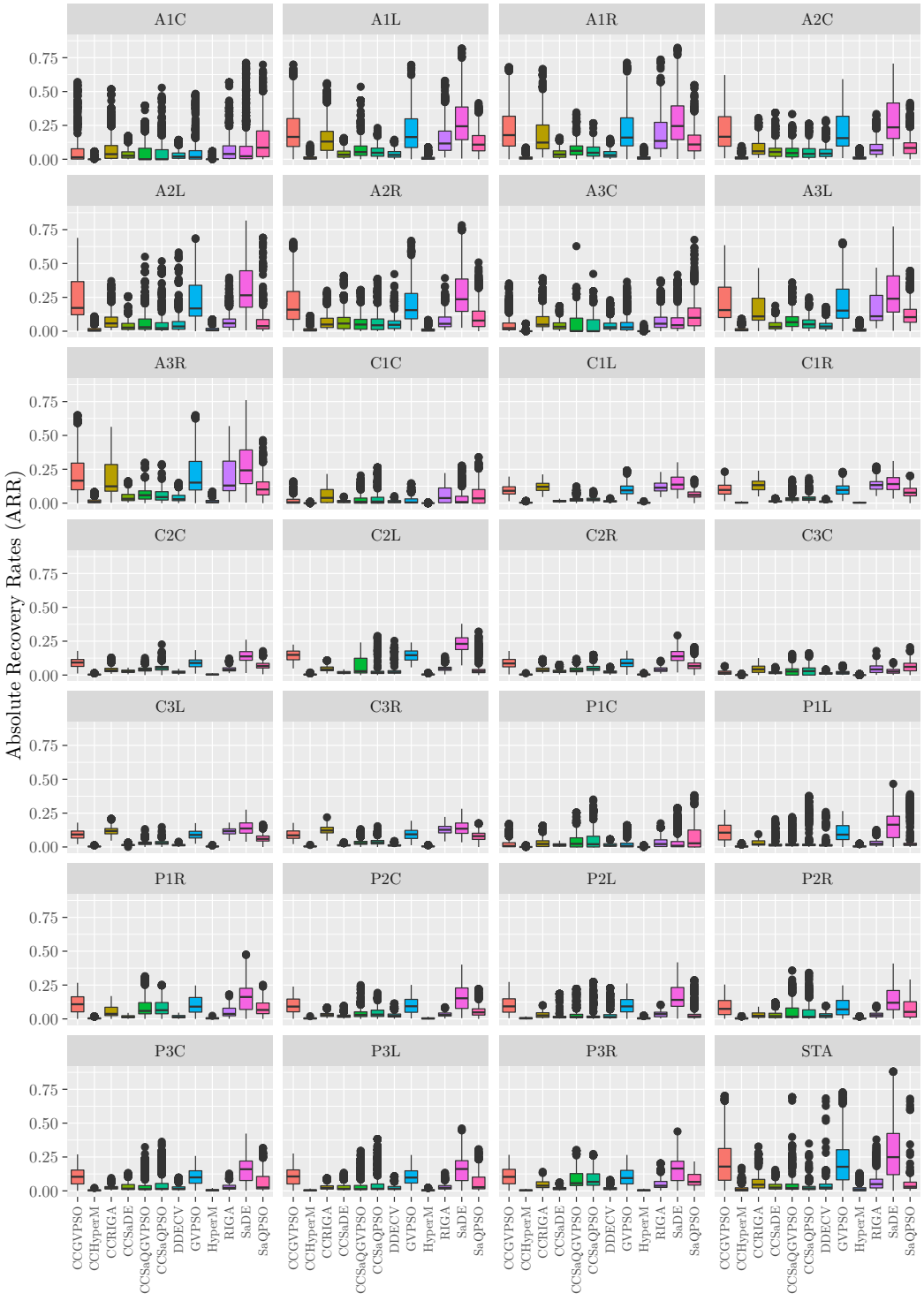


Figure 10.18: ARR of algorithms from the objective landscape perspective

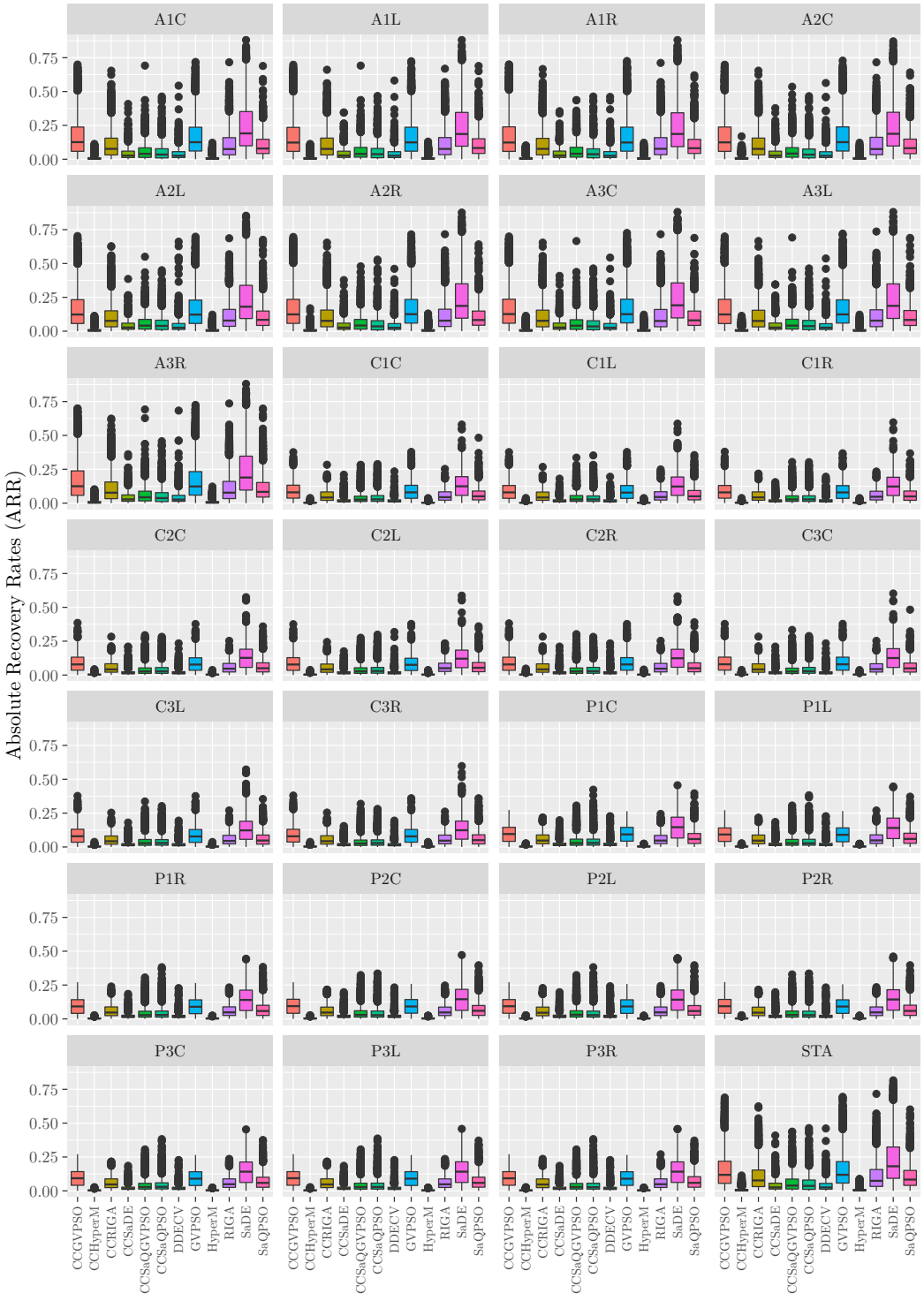


Figure 10.19: ARR of algorithms from the constraint landscape perspective

10.3.4 Overall Algorithm Recovery

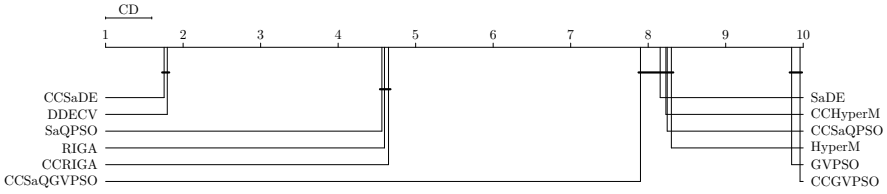
Figure 10.20 provides an overview of the optimisation algorithm recovery across the optimisation problem instances. For each of the recovery measures, namely P_{ABEBC}, P_{ABEAC} and P_{ARR}, a critical difference plot illustrates the algorithm performance preferences. From these critical difference plots, the same results are obtained that are previously observed, where CCSaDE and DDECV are undoubtedly the better performing algorithms for both P_{ABEBC} and P_{ABEAC} measurements, followed by SaQPSO. The dynamic co-evolutionary algorithms also generally present a slight preference over the non-coevolutionary variants. The critical difference plots indicate that no algorithm provided statistically significant performance based on the recovery measures. For the P_{ARR} critical difference plot, the performance of SaDE displayed the best improvement within change periods and is the best performing algorithm for this performance measure. As expected for the static optimisation problem within the optimisation problem change period, GVPSO and CCGVPSO display better recovery rates than the other algorithms, except for the performance of SaDE.

10.3.5 Recovery on Sampled Individual Problem Instances

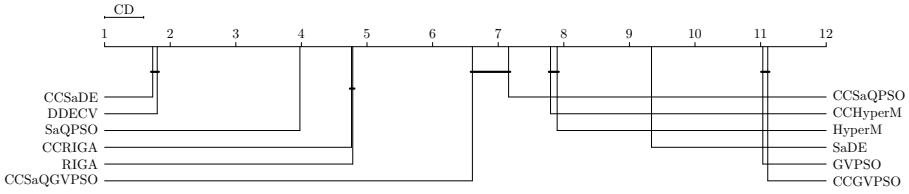
The sampled individual problem instances introduced in section 10.1.4 have their recovery profiles illustrated in figure 10.21 for the P_{ABEBC}, P_{ABEAC} and P_{ARR} measures. For each of the individual sampled problem instances, the optimisation algorithm recovery for the performance measures agree with the trends observed in sections 10.3.1 to 10.3.3.

For the P_{ABEBC} measure in figure 10.21a, CCSaDE and DDECV provided better P_{ABEBC} values for a large number of the total algorithm iterations within the `A1L/A1L` and `C3R/C3R` problem instances. The SaQPSO, CCSaQGVPSO and CCGVPSO initially provided larger P_{ABEBC} results but decayed to values lower than what the other algorithms managed to achieve. The remaining algorithms displayed fairly consistent P_{ABEBC} results for the `A1L/A1L` and `C3R/C3R` problems. Problem instance `P2C/P2C` illustrates that DDECV achieved the best P_{ABEBC} results, with more algorithms being able to increase their P_{ABEBC} values. Almost unchanged results are observed for CCHyperM, HyperM and SaDE, yet these results also indicate that the algorithms did not manage to provide any real improvements to possible found solutions.

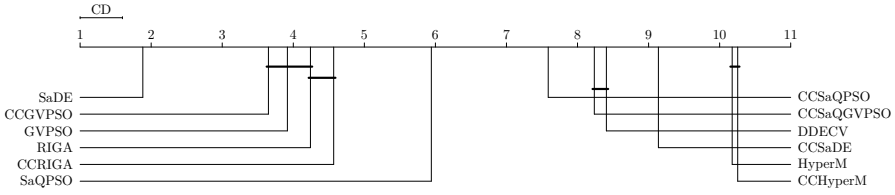
Figure 10.21b illustrates similar results for the P_{ABEAC} to that of the P_{ABEBC} measure. The main difference between the P_{ABEBC} and P_{ABEAC} results are that for the `A1L/A1L` and `C3R/C3R` problems, more algorithms displayed a drop in the P_{ABEAC} as the optimisation algorithm execution proceeded. This result implies that more of the algorithms did not maintain stable results after a change occurred in the optimisation problem. To be fair, this is not a completely unexpected result because the algorithms presenting a worse result



(a) Recovery based on the ABEBC measure

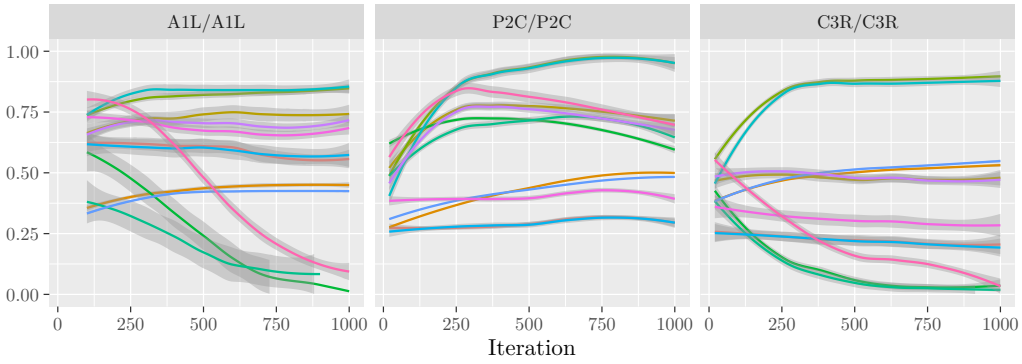


(b) Recovery based on the ABEAC measure

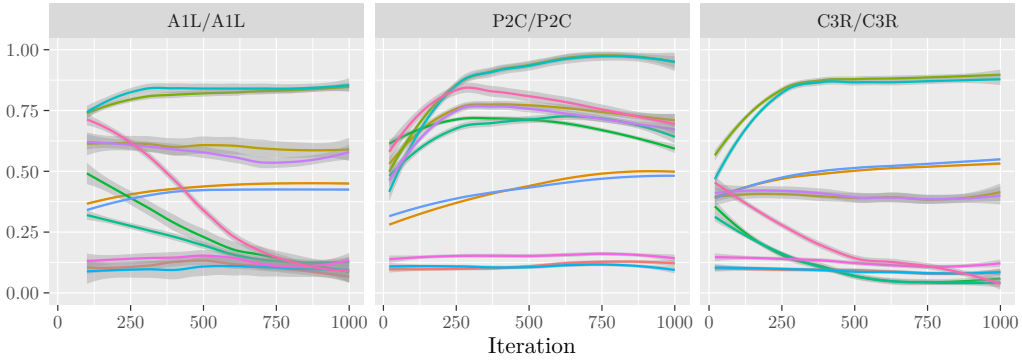


(c) Recovery based on the ARR measure

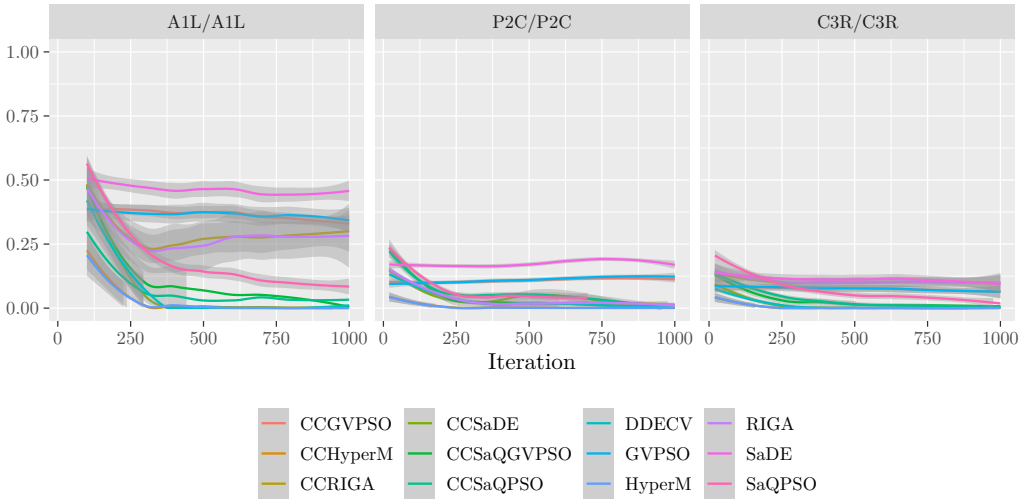
Figure 10.20: Critical difference plots of algorithm recovery measurements



(a) Algorithm recovery based on the ABEBC measure



(b) Algorithm recovery based on the ABEAC measure



(c) Algorithm recovery based on the ARR measure

Figure 10.21: Recovery of optimisation algorithms on individual problem instances

after the problem change also did not maintaining good solutions before the change to the optimisation problem. The P2C/P2C problem instance for the P_{ABEAC} measure also displayed similar results, whereby the SaDE, CCGVPSO and HyperM indicated lower P_{ABEAC} values.

Results for the P_{ARR} measure are provided in figure 10.21c and indicate that the best performance was by the SaDE algorithm. The other optimisation algorithms displayed reducing P_{ARR} results as the algorithm iterations increase. The performance results for CCHyperM and HyperM remained similar to that of the P_{ABEBC} and P_{ABEAC} measures, but are still worse than that of SaDE.

10.3.6 Conclusion

The results presented for algorithm recovery across the 784 unique DCOP problem instances indicate that the optimisation algorithms are able to provide solutions to the optimisation problems. The results presented for the P_{ABEBC}, P_{ABEAC} and P_{ARR} algorithm performance measures highlight that no single recovery measure is conclusive by itself and that multiple measures are needed to provide a more complete understanding of algorithm recovery performance. Recovery data supports the algorithm performance profile results within section 10.1, with a more clear representation of the recovery data provided within the critical difference plots in figure 10.20. From these critical difference plots, the same results are obtained as in section 10.1, where CCSaDE and DDECV are the better performing algorithms for both P_{ABEBC} and P_{ABEAC} measures. The critical difference plots also indicate that no algorithm provided statistically significant performance based on the recovery measures.

10.4 Analysis of Solution Feasibility

The feasibility of solutions within the optimisation algorithms is calculated as a simple percentage of the number of feasible solutions over the total number of candidate solutions. Performance profile vectors (see section 6.3) of the algorithm percentage feasibilty are considered to obtain a holistic view of performance across all algorithm iterations. The hypothetical ideal feasibility vector for an optimisation algorithm has all candidate solutions as feasible solutions. In other words, the hypothetical best feasibility for each algorithm iteration across optimisation problem changes would be 100%. However, such a feasibility vector is not likely nor practical. If the optimisation problem were to experience a change, a current infeasible candidate solution may become feasible, or a feasible candidate solution may become infeasible. Therefore, a balance of solution feasibility should be maintained not only to help the diversity of candidate solutions, but to also allow for better algorithm recovery after optimisation problem landscape changes.

This section begins with section 10.4.1 providing the feasibility percentages for the objective behaviour spaces, whilst the feasibility percentages for the constraint behaviour spaces are provided in section 10.4.2. The sampled individual problem instances from section 10.1.4 have their feasibility percentages discussed in section 10.4.3. Section 10.4.4 concludes this section.

10.4.1 Objective Space Behaviour

For the objective space behaviours provided in figure 10.22, a large percentage of the candidate solutions remain feasible through the execution of the optimisation algorithm. The profile plots across the algorithm iterations demonstrate that the feasible percentage of solutions never reduces below 80% when considering outlier results. Performance of the PSO algorithms are the most variable, however, this is not surprising because unlike with the GA and DE based algorithms, new genetic material is not continually being added to the candidate solutions of the PSO algorithms. The particles are instead re-evaluated and are encouraged to move into feasible problem search space through a particle update process. For example, the neutral particles of QPSO-based algorithms move through the problem search space by updating their velocity and position vectors respectively with equations (3.1) and (3.2). As a result of this particle movement, the percentage feasibility of SaQPSO displayed the largest interquartile range when compared to the other optimisation algorithms. Although a variance was observed across the iterations, the difference in percentage feasibility was at most 5% across the median results for SaQPSO.

The profile plots in figure 10.22 illustrate how the percentage feasibility drops at regular intervals. These intervals coincide with the changes experienced by the optimisation problem. An expected increase in the number of changes was observed for both the progressive and chaotic change behaviours within the percentage feasibility profiles.

10.4.2 Constraint Space Behaviour

The constraint behaviour space displayed similar feasibility percentages as the objective behaviour space. The PSO algorithms display the most variance and quantity of outliers whilst maintaining that at least 80% of the candidate solutions remain feasible.

From the perspective of the constraint behaviour space, more varied feasibility percentages are observed. The behaviour spaces with linear and random peak movement patterns produced observable fluctuations in the feasibilty percentages, particularly for HyperM, SaDE, CCHyperM and SaQPSO algorithms. The largest differences for these algorithms are observed within the `*2L` behaviour spaces, with the least amount of fluctuation observed in the `A1C`, `A3C`, `C1C`, C3C and `P1C` behaviour spaces. The most erratic and varied feasibility percentages are observed in the `C2L` behaviour space. Similar fluctuations are

observed for the A2L and P2L behaviour spaces. Moreover, this trend can be observed for all optimisation algorithms within these behaviour spaces.

As noted in section 10.2 the diversity of both HyperM and CCHyperM remain high throughout all behaviour spaces. The hyper mutation operator within these algorithms ensures that the diversity of the algorithm is maintained, but also allows for the possibility that infeasible candidate solutions are still possible as a result of the mutation operator. Furthermore, the poor recovery results (see section 10.3.6) obtained by both HyperM and CCHyperM also explain the varied percentage feasibility results because the algorithms are not able to sufficiently adapt before and after the optimisation problem experiences change.

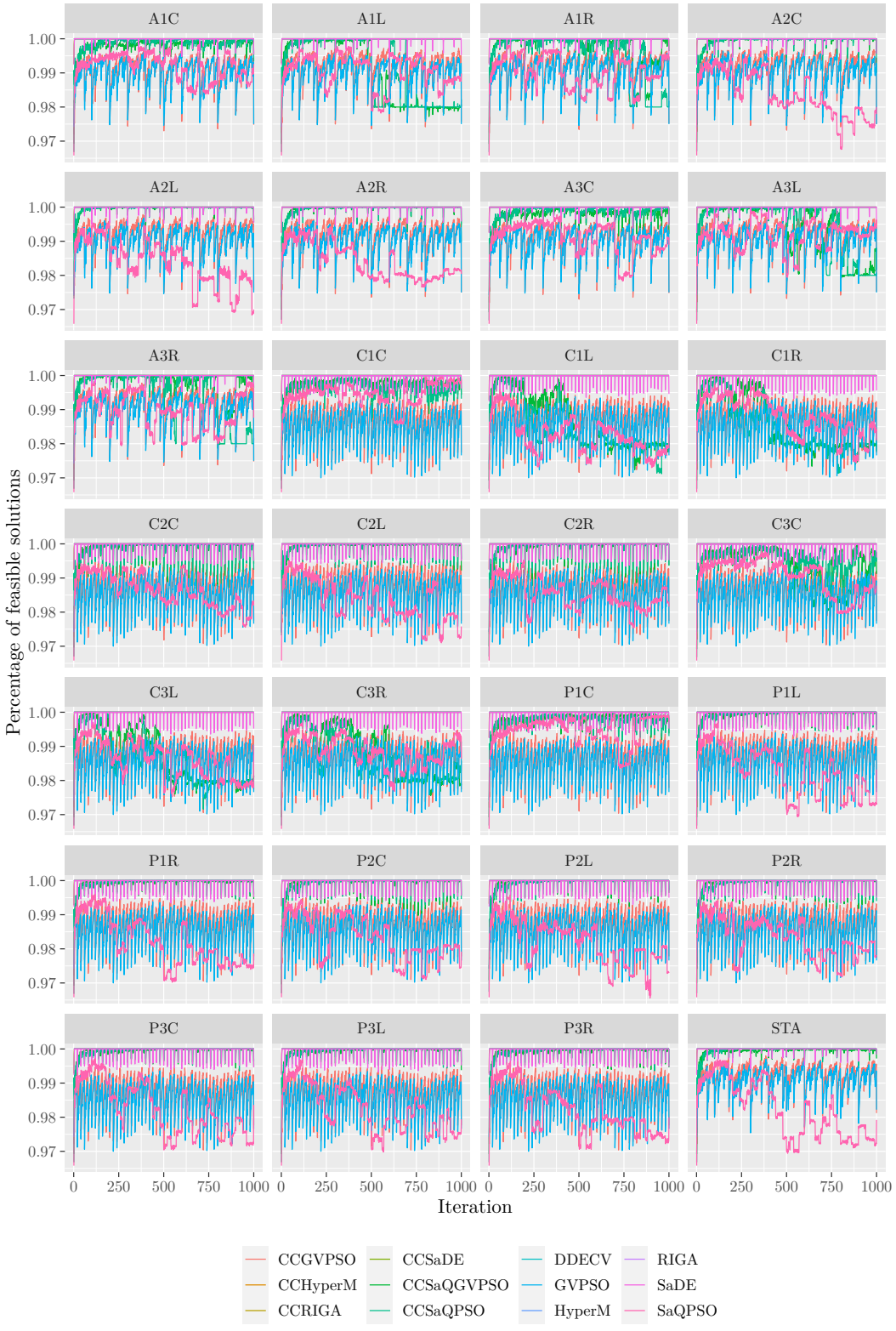


Figure 10.22: Feasibility percentage for the objective space perspective of constraint behaviour spaces

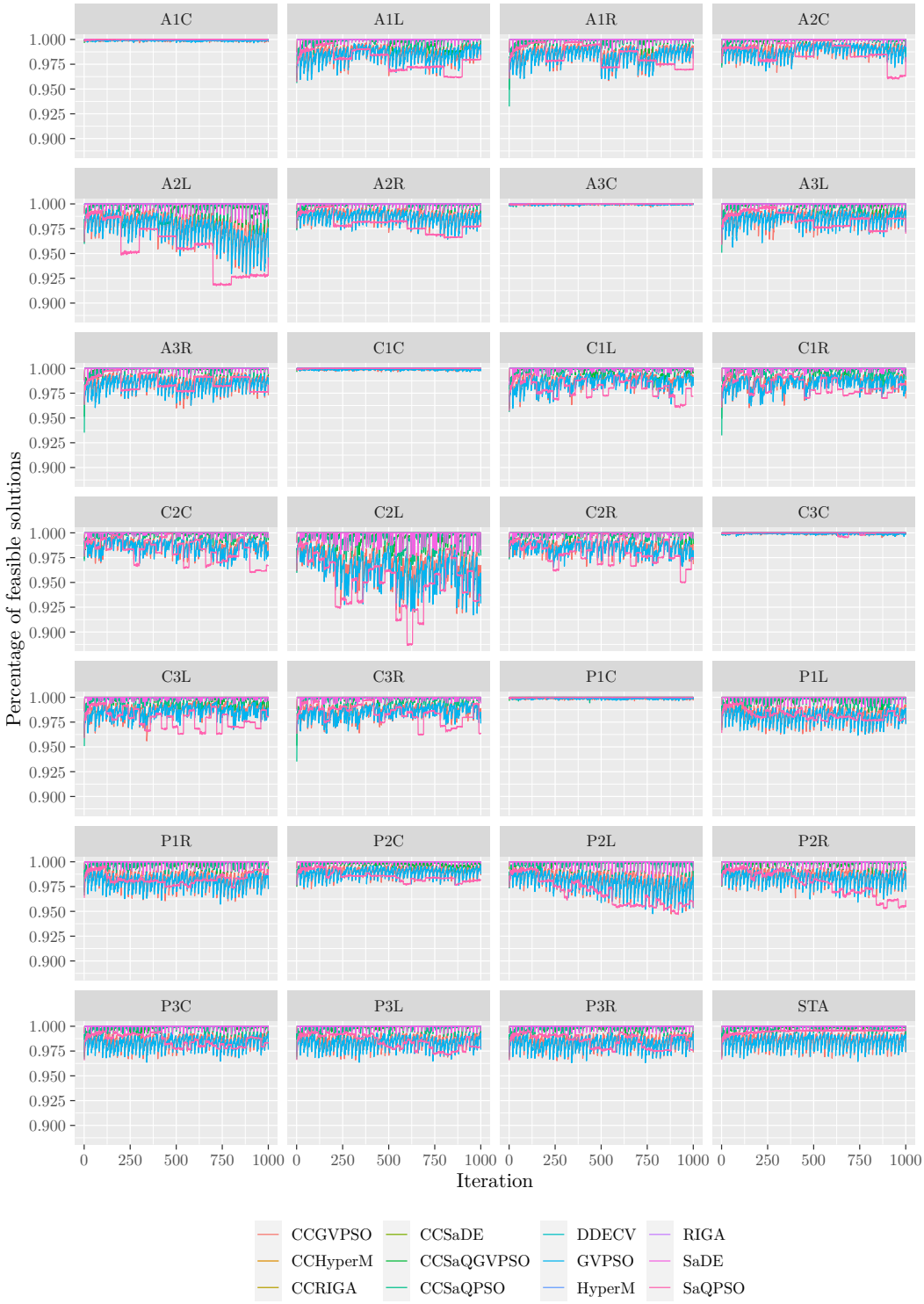


Figure 10.23: Feasibility percentage for the constraint space perspective of objective behaviour spaces

10.4.3 Feasibility Percentage of Sampled Individual Problem Instances

Solution feasibility for the sampled problem instances is illustrated in figure 10.24. From the feasibility information, the adjustments to the optimisation algorithms in order to cope with the change in the optimisation problem is clearly visible with the `A1L/A1L` problem instance. The `A1L/A1L` problem instance displays periodic jumps in the candidate solution feasibility which coincides with the changes to the optimisation problem. These periodic jumps are present with an increased frequency within the `P2C/P2C` and `C3R/C3R` problem instances which demonstrate increased problem complexity.

Across these individual problem instances it can be seen that the candidate solutions move towards feasible regions of the problem search space. For the `C3R/C3R` problem instance, the trend towards feasibility is far more erratic, because the problem instance experiences more changes than what are experienced for the `P2C/P2C` and `A1L/A1L` problem instances. Figure 10.24 also shows that, even though the candidate solutions are feasible solutions to the optimisation problem, the quality of the candidate solutions should be determined from accuracy measures and not only from the feasibility of solutions. However, lower accuracy results are obtained for algorithms that indicate more erratic feasibility percentages (see section 10.1). For the algorithms that obtained better performances in these problem instances (namely CCSaDE, DDECV, CCRIGA and RIGA) a more stable feasibility percentage was observed.

10.4.4 Conclusion

The candidate solution feasibility presented in this section for the 784 DCOP instances demonstrate that both the α-constraint and Lagrangian formulation are able to maintain feasible solutions. The results also indicate that the performance of the optimisation algorithm itself is more influential to the final results than the constraint handling method alone. Although the constraint handling does encourage feasible solutions for the optimisation problems, the ability for the optimisation algorithm to adapt and adjust to the changing problem landscape remains an important concern for DCOPs.

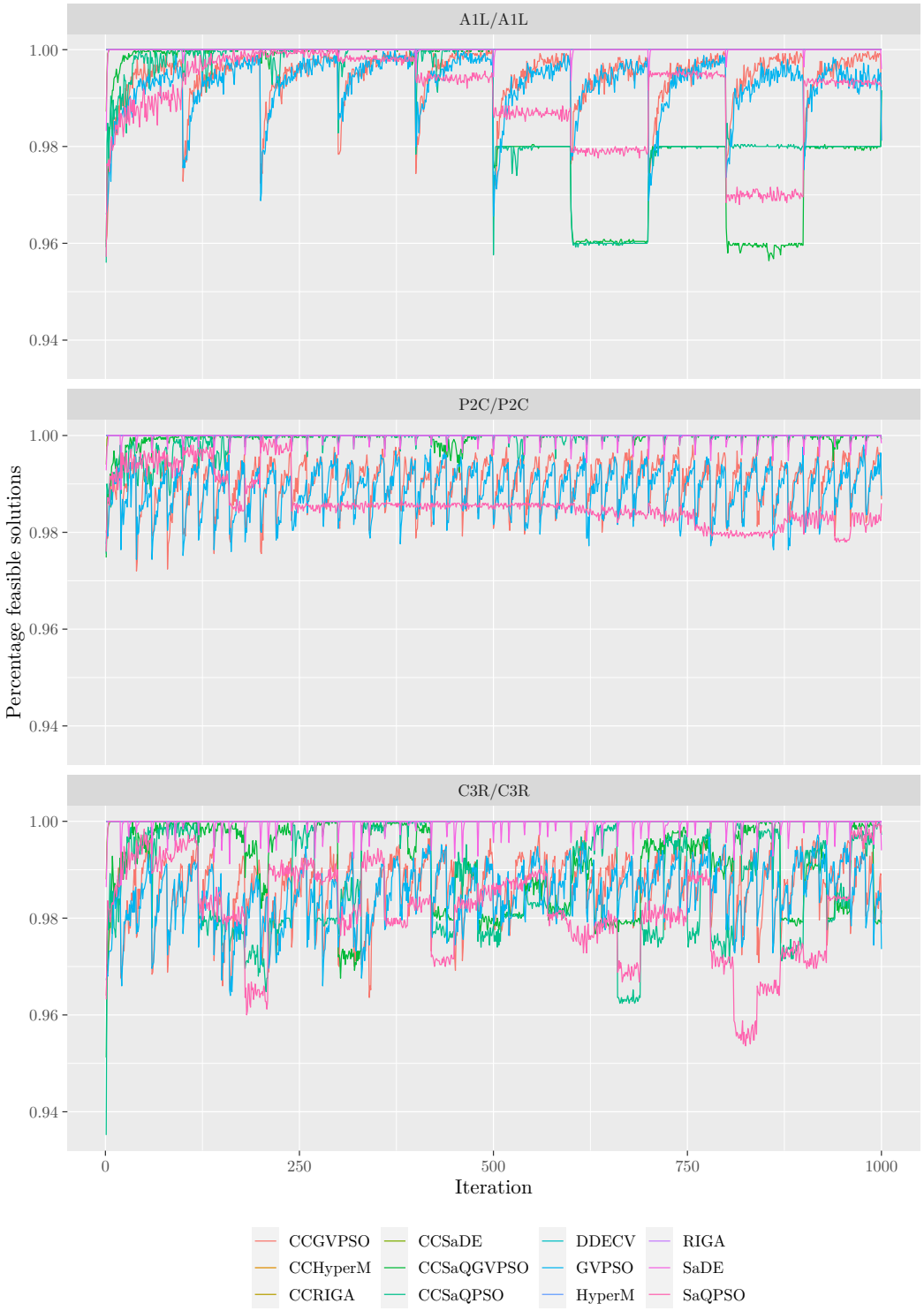


Figure 10.24: Feasibility percentage plots for the individual sampled problem instances

10.5 Conclusion

This chapter analysed the obtained results from the empirical process defined in chapter 9. The research questions raised in section 9.1 have been answered through the performance analysis of the dynamic co-evolutionary algorithms and the dynamic optimisation algorithms with α-constraint. The overall conclusions for the research questions are as follows:

1. *Can algorithms based on the dynamic co-evolutionary framework provide solutions to DCOPs?*

   The accuracy results provided in chapter 10 indicate that the dynamic co-evolutionary algorithms are able to provide solutions to DCOPs. In particular, the best performing algorithm was also the only algorithm to have achieved results that are statistically different, in a significant manner, from the rest of the optimisation algorithms. The best performance was obtained from the CCSaDE algorithm, which is a dynamic co-evolutionary algorithm. The other considered dynamic co-evolutionary algorithms displayed competitive results when compared to the dynamic optimisation algorithms with α-constraint. The dynamic co-evolutionary algorithms did provide solutions for all DCOP categories, namely SOSC, SODC, DOSC and DODC problem instances, as shown in section 10.1.3.1. It was observed for the SOSC and SODC categories of DCOPs that the SaQPSO algorithm achieved the best performance for all optimisation algorithms, but was followed by CCSaDE. In the DOSC and DODC categories, CCSaDE provided significant performance results for the DODC category of problem instances in particular. This is a significant result because the DODC category of DCOPs represents the majority of the problem instances within the benchmark problems.

2. *Does the choice of constraint handling approach matter?*

   From the accuracy performances of the optimisation algorithms it can be concluded that the choice of constraint handling approach does make a difference. For example, the accuracy performance results indicate that the SaDE algorithm using the α-constraint method provided the worst performances for the majority of the optimisation problem instances. The only exception was the only SOSC problem instance where SaDE achieved the best results, but was followed by CCSaDE. The poor performance of SaDE is overshadowed by the performance of CCSaDE, where problem constraints are handled using the Lagrangian formulation of the problem. Similar differences in behaviour can be seen between CCRIGA and RIGA with the α-constraint.

   Although the choice of constraint handling method does impact the performance of the optimisation algorithm, the performance of the opti-

misation algorithm itself has a far greater influence to the performance than the constraint handling method.

3. *Does the ratio of feasible to infeasible solutions reduce as the problem complexity increases?*

   The percentage of feasible candidate solutions in section 10.4 indicate that the number of feasible solutions does decrease as the complexity of the optimisation problem increases. This reduction in feasibility is particularly evident in the DODC category of DCOP problem instances. For the DODC problem instances, the progressive and chaotically changing problem instances demonstrate a larger variation in the feasibility percentage across algorithm iterations, for example, reductions of up to 10% are observed for the constraint behaviour space `C2L`. The `STA` problem instance provides an indication that the percentage of feasibility does increase for simpler DCOP categories in general. The CCHyperM and HyperM algorithms do, however, display lower feasibility percentages across the algorithm iterations, but this is explained through the process of diversity maintenance within the HyperM algorithm.

4. *Does candidate solution diversity provide an indication of algorithm performance for DCOPs?*

   The diversity analysis results presented in section 10.2 demonstrate that optimisation algorithms with larger diversity values do tend to have lower performances. Both the CCHyperM and HyperM algorithms provided mediocre accuracy performances whilst also being the only two algorithms to provide the largest consistent levels of diversity amongst the algorithms. Generally, the algorithms that maintained a slightly lower diversity level than that of CCHyperM, HyperM and SaDE provided better overall performance results. Therefore, candidate solution diversity remains an important consideration for DOPs as well as for DCOPs.

5. *Which approaches recover the best after the optimisation problem experiences a change?*

   Recovery performances for the optimisation algorithms were analysed in section 10.3. Figures 10.20a and 10.20b illustrate that the algorithms with the best performances are also the algorithms that displayed the best recovery when the optimisation problem changes. The best performances, in order, are from CCSaDE and DDECV, followed by SaQPSO, CCRIGA and RIGA. Even though different constraint handling methods are used by the better performing algorithms, the constraint handling method does not directly impact the algorithm performance. Perhaps it is more fair to rather highlight that the performance of the optimisation algorithm itself provides a far greater impact to the overall algorithm than what the

constraint handling method alone can do, particularly for more complex DCOPs.

6. *Are there differences in optimisation algorithm performance based on the category of DCOP instance?*

   The accuracy performance results in sections 10.1.3.1 and 10.1.3.2 show that algorithm preference differs based on the category of DCOP as well as the change type based on spatial and temporal severity. Simpler optimisation problem instances tend to favour dynamic optimisation algorithms whereas problem instances with an increased difficulty have more success with dynamic co-evolutionary algorithms. Furthermore, dynamic co-evolutionary algorithms tend to be the better performing algorithms for problem instances with larger spatial and temporal change severity.

From the answers obtained within the addressed research questions, it can be concluded that dynamic co-evolutionary algorithms are a viable option to solve DCOPs. Additionally, the choice of optimisation algorithm remains the most important choice for the optimisation process in totality because the optimisation algorithm determines candidate solutions. The constraint handling method remains important to influence the optimisation algorithm.

Part IV

Reproducible Computational Intelligence

Chapter 11

Reproducible Research

> *An article about computational science in a scientific publication is not the scholarship itself, it is merely advertising of the scholarship. The actual scholarship is the complete software development environment and the complete set of instructions which generated the figures.*
>
> D. Donoho

This chapter elaborates on the importance of applying the scientific method in order to affirm or disprove the theories presented by researchers, both within and outside of the field of CI. A discussion about the importance of reproducible research follows, including an overview of the currently observed process followed by practitioners. Popular software tools are examined and concerns about the tools are highlighted. The remainder of the chapter discusses the required fundamentals for software used within the scientific pursuits of the field of CI, followed by a discussion of a new software library which achieves the identified requirements.

11.1 Overview

As research continues within the field of CI, new algorithms and techniques are produced. The umbrella term of CI encompasses different research areas which include neural networks, evolutionary algorithms, swarm intelligence, fuzzy systems, and artificial immune systems [84]. The most prolifically observed CI research process can best be described as a "one-shot" or "once-off" culture, whereby experimentation and investigation are performed solely for the purposes of publication. Once results are published, the work performed to produce the research output is often forgotten. However, when further research into a subject is possible, the likelihood of forgetting or abandoning work is reduced. Researchers hope that the work which produced a publication would

have been archived in some form, so that it may be retrieved when required. Questioning, critiquing and verifying the work within a publication may provide answers to questions about unclear descriptions or about the process followed to achieve the reported results. Replication of published results is a critical part of the scientific method which may serve to confirm or debunk the presented results. The currently observed research process could be outlined as the following approximate process:

1. create and design a new process (algorithm or technique) that addresses a problem,

2. test the proposed process by evaluating its effectiveness on a set of benchmark problems and with competing algorithms and techniques that are accepted by the field as representative, and

3. analyse the obtained results and present them to the research community through a publication.

Unfortunately, reproducing data sets used within a publication requires a large time investment. The process is usually challenging when considering CI algorithms, where computations rely on non-deterministic or randomness to produce the data sets of solutions to an optimization problem. Although the random effects make it possible for algorithms to achieve performance goals, they simultaneously create unintended complexity, and through this complexity effectively prevent result duplication. Unintended omissions within the description of the algorithm implementation only serve to compound the difficulties associated with result data set duplication. Ultimately, the complexities of duplicating a data set precludes any algorithmic comparison described within a publication without access to the original program. The inability for comparison is however always present, even when the effects of non-determinism are ignored. Thankfully, other research areas of computer science, such as programming language theory, have developed representational schemes to express the representation of instructions and expressions for computer programs. Additionally, these representations are also verifiable by a computer language compiler thereby preventing invalid representations of expressions. The derived abstractions within mathematical logic create a specialized langauge which applies formal logic to mathematics in order to study the expressive power of formal systems as well as the deductice power of formal proof system. The abstractions present within the mathematical fields of *topology* [291], *category theory* [232] and *homotopy type theory* [281] in particular are directly applicable to computer science. By considering and including the improvements from mathematics and other research areas, the possibility exists to allow for more succinct problem representations and tools to reduce the inherent complexity within CI.

Removing the burden to recreate the design work of a publication in an attempt to reproduce its results is a very attractive proposition. The required time investment would reduce and allow for more interesting work instead of re-implementing the work presented within other publications. Previously, frameworks and libraries aiming to reduce this burden have been made available for use, often as free and open-source software. Open-source projects cannot remove all the complexity in implementing algorithms, but can make the process far simpler. The simplification of the process is usually due to the open-source projects focusing on reducing the time required to define an algorithm implementation. From these accelerated algorithm definitions new results sets may be produced far sooner than what would normally be the algorithm development process. Importantly, the improved speed of result production does not necessarily allow for the replication of results. Without the repeatable reproduction of results, the fundamental problem with the observed research process remains unchanged and possibly even more severe.

11.2 Importance of Reproducible Computational Intelligence

The reproduction of a study may be seen as being more than the sole reproduction of the initial data set. Data set reproduction should be possible, not only for the data, but also for the publication itself. Publications manipulate data sets and process the contained results to create summary statistics, which may be represented in tables or as graphs, or figures or plots. Solutions already exist to allow for the reproduction of a publication based on data sets. Literate programming [143] allows for the creation of documents where the data manipulation processes are included as part of the document itself. When creating the final publication document, the embedded logic within the document sources are evaluated, with the results becoming embedded into the publication document either as tables, figures, plots or as formatted output from the process itself. Importantly, the document creation process is reliant on *plain text* input documents which allow for simplified file management within version control systems. For example, the file formats produced by the more popular tools (such as Microsoft Word or Juputer Notebooks) maintain and produce files that tend to include binary data and is cumbersome for version control systems. Plain text input tools that are available for use which achieve this behaviour include:

- LaTeX documents with embedded `Sweave` [274] code blocks,
- Markdown or LaTeX documents with embedded `R` [274] code blocks where the documents are processed with a tool such as `knitr` [307–309], and

- `org-mode` [70, 253, 254] documents with embedded code blocks from different languages and tools.

Although literate programming documents fulfil the requirement to have the publication itself reproduced, other concerns should be considered in order to achieve a truly reproducible environment from which the publication may be produced. In the sections that follow, different data set representations are discussed in section 11.2.1 which are the output from reproducible optimization algorithms. The challenge of reproducible optimization algorithms is discussed in section 11.2.2 with a complete reproducible execution environment for reproducible research discussed in section 11.2.3.

11.2.1 Data Set Formats

Without the experimental data it is not possible to derive a publication. As a result, the value of a data set cannot be overstated. Another equally important concern is the file structure of the data set itself. In this case, the referred to structure is the file format of the data set. The most common formats for a data set are based on plain text files. Plain text files include formats such as comma separated values (CSV) text or encoded data formats such as JavaScript Object Notation (JSON), which is persisted onto a storage media as plain text. Although such data representations are valid, the disadvantages associated with the use of plain text files negatively impacts the tools that consume the data to derive information. The disadvantages for plain text formats include:

- **File size:** To represent data within a plain text file, all data types are converted into strings. The string based representation results in large files which may become cumbersome to use. Large text files may also necessitate the use of external applications in order to compress the file contents. Smaller sized files are simpler to transfer and allows for more efficient use of storage media. Formats such as JSON only add to the size of the file by enforcing that all data be encoded into a structure defined by a grammar within the plain text representation.

- **Data orientation and inefficient querying:** The data items for plain text formats have a row based orientation. Each data record is written to the file as a contiguous block of text. For example, extracting the n-th column from a CSV file requires reading and/or passing over all $n-1$ preceding columns, for each row within the file. During the analysis of data, columns are often considered in isolation in order to calculate derived values, such as summary statistics.

- **Absence of schema:** Textual formats may only represent a textual type of data. The implication is that during data processing, the data

within the file must be interrogated in order to determine the types of data that the textual items represent. The interrogation process usually requires a parser program which attempts to guess the correct data type for a piece of data. Parsing is a slow process which requires a percentage of the data file to be processed (at a minimum) in order to assign data types to the data. Alternatively, a user may explicitly tell the parser what data types are expected. This explicit process has the drawback that the user should instead interrogate the data in some manner to determine the expected data types if they are not already known.

Extensions to plain text data formats have been developed in attempts to alleviate the need to assign data types during the paring process. For example, the specification for JSON-Schema [135] attempts to add a schema definition to JSON data, requiring a query of the schema to determine the expected data types. The defined schema is far smaller in size than the actual data and dramatically reduces the interrogation processing time of the contained data. Although a schema does remove the problem of data type assignment, the data items are still stored in a row-orientated manner which results in wasted processing time when a single column value is required. To improve the speed of data access, a data set may be stored in the running memory of a computer. The data in memory may still prove to be too large, possibly necessitating "on the fly" compression. Dynamic compression techniques may prove to be effective but as the size of the data set increases, the techniques may fail to scale efficiently.

Columnar storage data formats achieve solutions to most of the previously mentioned disadvantages to row-orientated data formats. Columnar formats allow for efficient retrieval of a given column of data values and may also allow for compression of the columnar data because the same type of data is contained within a single column. Columnar data files are akin to a table within a database:

- data values may be accessed directly;
- columns of data types may be compressed using a compression scheme suited to the type of data within the column;
- data can be more efficiently stored using binary representations instead of plain text;
- file sizes are reduced due to the use of binary representations and compression; and
- the schema for the data is implicitly available without additional processing.

As data sets grow in size and become more common place, particularly with machine learning algorithms becoming more popular, several proposals for binary columnar data set formats have been made. Importantly, the data file format should be an open-source format. By defining the format as an open-source format, analysis tools are generally more amenable to integrate such formats. Better integrations with analysis tools allows for more choice by the user of the analysis tool.

The current recommended format is the *parquet* [5, 188] file format. Parquet files are already accepted by most data analysis systems and have the additional benefit of having comparatively small file sizes. For this reason, the *parquet* format is recommended to be the container for experimental results from CI algorithms. Providing an open-source data file format allows for simplified access to data without necessitating the use of specialized software tools as part of the analysis process to produce a publication document. Efforts to simplify the interoperability in creating and manipulating parquet files include the *Apache Arrow* [4] project. The *Apache Arrow* project aims to provide zero-copy reads, fast data access and interchange of data within parquet files but without incurring large serialization overheads during the interchange. For example, the data transfer process between the Spark [314] platform and `R` [274] demonstrates a speed-up factor of 16 when using Apache Arrow as the memory interchange format. The use of open-source should not only be restricted to data file formats but also to any tool or software used to produce analysis results of data for publication. Open-source software also allows for simpler review and inspection of the logic used to produce a data set.

11.2.2 Stochastic Optimisation Algorithms

Algorithms within the fields of evolutionary computation (EC) and swarm intelligence (SI) are all stochastic algorithms. Stochastic algorithms attempt to provide solutions to an optimisation problem through the use of random variables. For many, if not all, of the algorithms within EC and SI, the algorithmic model is based on a metaphor of biological systems appearing in nature. These algorithms are also known as "nature-inspired" algorithms.

Regardless of the internal mechanisms of an algorithm, the use of randomness during the optimisation process is required. Computation on a computer system is, however, a deterministic process. Computer systems use the current state of the computer memory and user inputs (among other information sources) to generate entropy. Such a source of random information could be sampled within an optimization algorithm to generate random variables. Unfortunately, such sources of entropy are not always reliable and the possibility exists for the source to become exhausted.

To address the possible loss of entropy, computers instead make use of a deterministic algorithm to generate a pseudo-random stream of information. The pseudo-randomness is initialized using an initial *seed* value. The purpose

of the pseudo-random generator algorithm is to produce a stream of "random" values such that no statistical test, within a class of statistical tests, can observe any difference between the generator output and a uniform distribution. By using statistical tests, the pseudo-random number generator (PRNG) algorithms may be ranked based on the perceived strength of the generator, the numerical range and period of the generator, or even on the computational complexity of the PRNG algorithm itself. From the PRNG ranking, an appropriate generator may then be used within a computer program.

For the sake of convenience, most programming languages provide an interface to the programmer to obtain random values. The choice of generator selected for this task has previously been a contentious issue. For example, the default generator available for use within the C programming language (the `rand` function, which implements a variation of the linear congruential generator (LCG) algorithm) has been shown to provide a lower level of statistical randomness contained within the lower set of generated bit values. The recommended usage is to only sample the upper set of bits for the produced random value, or to alternatively simply use a different generator in order to work around the problem.

The fundamental problem with implicitly provided PRNGs is that they are seeded by either the operating system or the programming language runtime and do not make the initial seed value known. The initial seed is often, but not always, determined during the boot process of the operating system. As a result, any data obtained through the use of an optimization algorithm is not repeatable nor reproducible. Therefore, knowledge and/or control of the PRNG seed value and the generator itself should be foundational to create reproducible results. When sampling a value from a PRNG, not only is a random value obtained, but the internal state of the PRNG is modified in-place. The state modification is required to allow the PRNG to produce the next value in the random stream on a subsequent invocation.

11.2.3 Reproducible Simulation Environments

Several attempts to allow for reproducible research have been pursued with different degrees of success. The current trend is to control and manage the development environment for the research using Docker [190, 205]. Docker is a containerisation technology allowing for sandboxed runtime environments which can be prevented from accessing the host system and possibly the network and internet. Docker usage produces different problems during execution, although it is relatively successful at allowing for a standardised starting point.

Once the docker container is running, it is possible for the user to mutate the underlying system within the container. These mutations seem harmless but may eventually become serious impediments. The "layers" that docker provisions on the host system are not isolated and may be altered from outside of the container image. Changes applied within the docker container also

result in an altered set of data layers (specifically in the last layer). From the base system image provided by docker, the user may require additional tools or libraries to be included within a derived container. Such additions are often managed through a package manager and are often retrieved from the internet using a weak matching system such as `SemVer` [236]. `SemVer` does not afford any guarantees about the quality and the state of the software that is represented by the current `SemVer` version value. The premise with `SemVer` is that valid `SemVer` values should be used and assigned by software maintainers based on a set of agreed upon rules within the `SemVer` specification document. The reliance on maintainers to select a representative and correct `SemVer` value is based on trust. Unfortunately, the maintainers of projects are human and mistakes are often made which completely invalidates the intention and effectiveness of `SemVer`. If the version number assigned to a software release were to be validated by an automatic process instead, more confidence could be placed on the `SemVer` value.

The ideal scenario would be a system that affords the ability to define not only the initial base runtime, but *any* additional required software as well. The additional software may be external tools and/or libraries and should form part of the definition for the runtime environment. Docker simply does not allow for these guarantees. It is possible to obtain a different docker container from the same set of inputs because the tooling used during the container creation is itself not able to consistently produce the same output result.

A possible solution which prevents all the previously mentioned concerns is the `nix` deployment system [69]. `nix` is primarily a tool to describe a declarative, pure, functional language which provides the foundation for a package manager. Using `nix` it is not only possible to define the base runtime system, such as a complete Linux installation, but to also declare any included libraries and tools to produce a fully reproducible deployment.

`nix` is a source-based package manager which considers the source code for a package as the absolute truth to specify the package's dependencies and version. As previously mentioned `nix` is a pure, functional language and the core abstraction is the mathematical function. Therefore, the inputs to a package definition within the `nix` language are the project source code and any additional inputs required to successfully build the given project. Within `nix`, package declarations are simply called "expressions", or more specifically "nix expressions". Consider the beginner `C` program that produces the console output of `Hello, World!`. The `hello-world` program requires the `main.c` file (containing the entry point of the program), together with a `C` compiler and standard library. The aggregation of the required inputs to build the `hello-world` program can uniquely define a cryptographic hash and this hash serves as input to the `nix` language expression. The purity property afforded by `nix` guarantees that any changes in the input to the expression will result in a change to the output. Furthermore, these ordering guarantees for the input allow for the caching or memoisation [193] of expression outputs. Given

a specific input hash, the corresponding output may be provided as an already built artefact instead of rebuilding the package once again.

Although possibly slower to create an environment without any precached `nix` expressions when compared to Docker, the environment produced by `nix` is truly reproducible and does not take any underlying state into consideration.

11.3 Current Tools

Before investigating possible solutions to manage and control the PRNG usage problem mentioned in the previous section, other available software tools for EC and/or SI are first discussed. These tools have previously been used for EC and SI research with the outputs being referenced within publications. The list of currently available and popular software tools include the Python programming language and its available ecosystem of libraries, Matlab, Waikato Environment for Knowledge Analysis (WEKA), ECJ, jMetal, Computational intelligence library (CIlib), ecr and Library for Evolutionary Algorithms in Python (LEAP). Finally, currently available tools for deep learning neural networks are discussed.

11.3.1 Python

The Python programming language has become popular within the research area of data science due to the large number of available software libraries. This popularity is largely due to the simple language syntax and the perceived simplicity of the development experience. When using Python, the following must be taken into account:

- The language is strongly and dynamically typed.
- Python is an interpreted language and imposes an evaluation tax on program execution.
- When considering execution performance, libraries are often written in lower-level, optimized languages when the performance of pure python becomes problematic. The configuration and usage of the language foreign function interface (FFI) bindings to these optimized libraries may become a source of errors which are troublesome to solve.
- Python may be regarded as a tool to glue other tools together.
- Python does not have mature package management solutions. Problems may arise due to version conflicts between versions of libraries and the programming language itself.

Dynamically typed languages are often presented as extremely versatile languages which bend to the will of the programmer. Unfortunately, the versatility afforded by dynamic languages often produces problematic program execution behaviour. Dynamic languages allow for the redefinition of functions, methods, classes and variables at any stage during a program's execution. Although it is possible to work around any identified problem within a library by using such redefinition, it does raise the question of whether any guarantees for a reproducible program execution are possible. Tools to statically validate Python programs before execution do exist [203], however the effectiveness of these tools is often limited to the current project source code and does not necessarily extend to the included project dependencies. As a result, it is generally not possible to have the confidence that a Python program (in totality) is actually reproducible.

The most popular libraries available for scientific computing in Python include `NumPy` and `SciPy`, with `Keras` and `PyTorch` extending these libraries to interface with deep neural network frameworks. Examples of libraries with focus on evolutionary algorithms are `LEAP` and `Nevergrad`.

11.3.1.1 NumPy and SciPy

From inception Python did not have support for any form of comprehensive numerical computation, except for the use of basic primitive mathematical operations. An extension library called `Numeric` [195] aimed to address these shortcomings but was eventually replaced by the more optimized `Numarray` [295] library. `NumPy` [75, 217] was eventually released as the successor of both `Numeric` and `Numarray` which unified both libraries and added a series of improvements. `NumPy` provides faster vector and matrix based operations, but is limited to these data structures. These operations are represented by the `ndarray` data structure which is almost entirely implemented in the `C` programming language for the performance reasons previously mentioned. Scalar operations are not part of `NumPy`, relying on Python to perform any scalar value operations. As a result, any operation that may be represented as a vectorized operation can be efficiently implemented using `NumPy`.

`SciPy` is not a library for scientific computation by itself, but is instead an umbrella project with the goal of enabling scientific computing in Python. Using `NumPy` as the base library, `SciPy` expands the set of available operations by integrating several other software tools together using a standardised interface. The extensions include modules for optimization, linear algebra, signal and image processing as well as Fourier transformations and special functions.

11.3.1.2 Keras and PyTorch

Deep neural networks have become a popular area of research in recent years. The `Keras` and `PyTorch` [228] libraries implement bindings for deep neural

networks on top of the `Tensorflow` [185], `CNTK` [255] or `Theano` [276] toolkit frameworks. Other toolkit frameworks are also available but are not as popular nor as complete. Instead of being part of the `SciPy` project itself, these toolkit frameworks are often used to complement the functionality afforded by `SciPy`. As with the concerns raised about the reproducibility of computations in Python, the extension frameworks like `Tensorflow` only serve to compound the problem by offloading the training process of deep neural networks. Although the central processing unit (CPU) can be used to train networks it is common practice to offload the training to the graphics processing unit (GPU). A GPU is a specialized piece of hardware that is purpose built to render graphics from in-memory scenes onto a display (such as a computer monitor). Although GPU processes do provide a significant reduction to the time required to train a model they also impose several restrictions. The programming interfaces to the GPU do not allow for the control of randomness sources which makes reproducibility impossible and hardware dependent. Furthermore, due to the cost of bandwidth and memory within a GPU the precision required within a GPU is often less than that available on a general computer CPU. Depending on the scenario, the reduction in precision may not be a problem. For scientific computation the reduction in precision may result in unacceptable errors being introduced during the process which may begin to snowball out of control, with the final result potentially being unacceptable.

11.3.1.3 Library for Evolutionary Algorithms in Python

Library for Evolutionary Algorithms in Python (LEAP) [49] describes a software library implementing multiple evolutionary optimisation algorithms. The intention of LEAP was to make the use of algorithms within research contexts simpler, whilst allowing for industry application and simpler study by students. LEAP is primarily focused on EC algorithms.

LEAP describes the flow of an optimisation algorithm through the use of a pipeline metaphor. As candidate solutions enter the pipeline, multiple transformations are applied before the result replaces the original candidate solution. LEAP is not immune to the problems of `Python` and additionally uses the standard PRNG provided by the language. As a result, the resulting algorithms are not reproducible.

11.3.1.4 Nevergrad

`Nevergrad` [240] is a Python library implementing different evolutionary optimisation algorithms using `NumPy` as a basis. The library is still immature and is seen as a complementary project to `PyTorch`, `Tensorflow`, *etc.* Because the library is an extension of `NumPy`, all the previously mentioned problems associated with it and generally with Python are also present within `Nevergrad`.

11.3.2 Matlab

Matlab is software which provides a multi-paradigm numerical computing environment and programming language. The software is proprietary, developed by MathWorks and requires the purchasing of a usage license. Matlab was originally designed to operate on and manipulate matrices but has expanded its focus over time. The current feature set of Matlab includes:

1. A general object-orientated programming language for use within the toolkit
2. Matrix manipulations and operations
3. Ability to plot mathematical functions and data
4. Simulation and implementation of algorithms
5. Creation of user interfaces
6. Interfacing with other programs written in other programming languages

For scientific computing, Matlab provides a single environment within which a user can perform a variety of operations and analyse data for inclusion within a publication. The toolkit also has a flourishing extension ecosystem where users may publish their packages for other users to use. Matlab does, however, share the shortcomings of Python mentioned in the previous section and is also not as scriptable as what Python is, hindering the usefulness of the platform. Notably, version conflicts can occur frequently between the toolkit platform itself as well as within user extension libraries. It is recommended that the exact version of Matlab be cited within a publication to limit possible platform incompatibilities. Matlab also implicitly provides the user with a seeded source of randomness which is cumbersome to control. As a result, although the toolkit framework is useful it does provide significant impediments for reproducible research. Alternatives such as Julia [15] and Octave [80] may possibly address the shortcomings of Matlab scripting and to bypass the expensive licensing cost, whilst providing a similar user experience. Unfortunately, the Matlab alternatives do not have enough extension libraries available or simply have a small user base.

11.3.3 WEKA

The Waikato Environment for Knowledge Analysis (WEKA) [94, 110] project provides a collection of visualization tools and algorithms for data analysis, model prediction and data mining. The toolkit provides the user with a graphical user interface (GUI) to allow access to the provided algorithms and functions for data preprocessing, clustering, classification, and feature selection.

WEKA additionally also provides an application programming interface (API) for programmatic usage of the toolkit. Within the API, the user is expected to provide a custom PRNG object where the appropriate initial seed value has been set. Even though the API does make provision for a custom instance of the PRNG to be provided, the interface for the generator is provided by the standard library of the Java programming language. The initial seed value for the PRNG is expected to be the concern of the toolkit user, with the project documentation describing what the expected usage of the PRNG to WEKA should be.

Due to the scope of WEKA being predominately focused on visualization and data mining, the use of WEKA for EC and SI is not recommended.

11.3.4 ECJ

ECJ [173] is a toolkit for evolutionary computation written in Java. The toolkit takes a component based approach to algorithm specification, allowing different parts of a basic evolutionary algorithm to be exchanged in order to produce different algorithm implementations.

The core of the library defines a series of inheritance hierarchies for each component within the general EC algorithm. Although possible to use the toolkit as a software library in a programmatic fashion, the preferred usage is through a command line interface (CLI) `Evolve` application, which is part of the toolkit release. The `Evolve` application defines a process which reads a set of hierarchical configuration files (based on a key-value format) to define the structure and configuration of an EC algorithm. Extra functionality such as a GUI based application with plotting support is also available. Specialised algorithm implementations are already defined and available, with extensions for network communication and distribution.

In order to allow for the parameter file configurations, the components are dynamically instantiated at runtime through the use of Java reflection and introspection. As a result, this configuration process does not detect invalid algorithm definitions but instead relies on the `Evolve` application to fail when a configuration is invalid. It should be noted that there exists the possibility to not have a configuration fail at runtime, yet be a logically invalid algorithm definition. When results are obtained for an invalid algorithm configuration it is hoped that the results are not published as valid.

The style of configuration employed by ECJ, together with all the additional features not directly related to the algorithmics in EC, results in complicated algorithm definition process. ECJ has a large learning curve as a result of the toolkit complexity, necessitating user documentation to specify configurations that are valid and to highlight usages that may be problematic. Although the documentation is extensive, the documentation is not verified against the project source code. Without ensuring that examples in the manual are valid

and working, large portions of the manual may potentially be invalid which will cause confusion amongst users with regards to toolkit usage.

11.3.5 jMetal

jMetal [78, 79] is a Java-based, object-orientated framework for multi-objective optimization with meta-heuristics. Unlike the configuration file approach followed within ECJ, jMetal provides the user with a set of predefined algorithm templates. The algorithms may have different algorithm components replaced as long as the replacement adheres to a defined interface provided within the framework. Compared to ECJ, jMetal has a superior verification process regarding algorithm definition whilst also presenting as a component based framework. Algorithms are defined using Java source code and may be checked by the compiler before execution, at the cost of a potentially longer algorithm configuration time-period. With the enhanced verification of algorithms, it is still possible to produce algorithm implementations that are logically invalid. To detect and diagnose that implementations are logically invalid, knowledge of how the internals of the jMetal framework operates may be necessary.

The framework provides a general algorithm runner abstraction with which all the defined algorithms are compliant. The "runner" allows for a single point of entry in order to use the framework. jMetal is a framework focused on multi-objective optimization and whilst providing similar functionality to ECJ, the focus of jMetal remains niche. Although documentation for the project does exist, the documentation focuses on the object-level structuring of the framework itself. From this object-level structure, automatic documentation may be generated (known as *Javadoc* [220]) for the user to study in order to familiarize themselves with the framework.

11.3.6 CIlib 1.x (Java version)

CIlib [42, 223, 229–231] is a Java based framework toolkit for population-based optimization algorithms, including EC and SI algorithms. Developed independently from ECJ, the toolkit developed a similar process to declare and execute algorithms. The notable exception is that the configuration is not hierarchical. Instead, the configuration is based on an XML algorithm specification. The XML-based specification defines the algorithm, the optimization problem, required measurements and data output together with the data output file-format.

Similarly to both ECJ and jMetal, CIlib 1.x declared algorithms using a component based approach. The problems highlighted with ECJ regarding algorithm validation and runtime failure also exist in CIlib 1.x. The reflection-based algorithm creation process took place whilst parsing of the XML algorithm specification.

The component hierarchies resulted in an inter-connectedness of program logic and objects. Unfortunately, this inter-connectedness was the source of many unexpected defects, as well as reinforcing the inability to correctly create reproducible results. Furthermore, more user documentation was required to explain the workings of the toolkit because the reflective algorithm process was not obvious and error-prone.

It was eventually decided that the manner in which CIlib 1.x operated was not conducive for research purposes. The learning curve required to use the framework and the complexities in extending the toolkit framework were an impediment to users. A new formulation of the CIlib library which deprecates the CIlib 1.x implementation and addresses the identified problems within current framework toolkits and libraries, is discussed in detail within chapter 12.

11.3.7 Evolutionary Computation in R

`ecr` [21] is a library for evolutionary computation within the `R` programming language. `R` is specialised to statistical programming applications and encourages the use of normal functions together with function composition. The library specifies a predefined algorithm structure for EA algorithms with the opportunity for the user to override individual algorithm steps within the algorithm structure. Essentially, the library allows different user defined functions to "plug into" the defined algorithm template.

Due to `R` having a dynamic type-system, functions and data values can be altered at any point within a program's execution. As a result, the same concerns that were highlighted with Python (see section 11.3.1) are also applicable to `R` and to `ecr` by extension. Any functions that define an incorrect set of parameters, access missing data or calculate invalid results due to different expected data types are only found at execution time, allowing for the incorrect usage of the library. Furthermore, the usage of PRNG instances is not well-defined, nor controlled within `R`. As a result, the resulting algorithm definitions are able to provide reproducible results from their execution.

11.3.8 Paradiseo

The Paradiseo [137] framework has the aim of allowing the user to quickly assemble an evolutionary optimisation algorithm. The framework is designed in a component-based manner, allowing the user to replace compatible components to alter the behaviour of the resulting algorithm. The framework is implemented using `C++` and claims to be the fastest available execution framework for evolutionary algorithms.

Paradiseo is broadly divided into four main components. The primary component is known as the `core` module, upon which the remaining modules depend. The remaining components extend the functionality of `core` by provid-

ing more algorithm implementations for single and multi-objective algorithms and lastly by providing concurrency and hybridisation.

Unlike other frameworks (such as ECJ and jMetal), Paradiseo provides no runtime configuration of the algorithm and requires that the user specify the algorithm implementation at compile-time. Although troublesome in terms of program preparation, multiple programming errors can be prevented using the assistance of the `C++` compiler with its template generation capabilities. A counter-point to this benefit would be that it is possibly more difficult to detect and determine the source of possible errors because the framework allows for the in-place mutation of data values by default. In-place mutation is fast: the combination of the operating system and running program need not concern themselves with the allocation and de-allocation of additional memory by changing already existing allocated memory. This "speed" does, however, come at the potential price of correctness.

The in-place memory adjustments made within Paradiseo impose a further problem because the PRNG within the framework is not thread-safe. Without the guarantee that the access to the common, shared PRNG instance is protected, the order of PRNG sampling cannot be enforced. Because of the inherent problems associated with concurrency, individual runtime processes are instead preferred thereby limiting the exposure of the PRNG. Following this design decision, the parallelism functionality provided by Paradiseo exploits execution specifications such as `OpenMP` [53].

Overall, the framework provided by Paradiseo allows for the definition of evolutionary algorithms. The produced algorithm may provide good execution speeds (provided the user does not introduce a slowdown), but is still quite primitive in regards to the complete functionality provided to the user. Output data is not clearly defined nor the way to persist this information for later analysis, shifting the problem to the user. This is contrary to the expected functionality provided by a framework, whilst simultaneously preventing reproducibility.

11.3.9 Deep Learning Tools

As previously mentioned in section 11.3.1, a number of tools are currently available to allow the use of deep neural networks. Although these tools are focused on deep neural networks, the caveats already mentioned about Python are inherited by these tools. In addition to `PyTorch` and `Keras`, the following are also available

- `Caffe` [129] is a deep learning neural network project that is particularly focused on applications of computer vision. A large portion of the project is implemented in `C++` in order to improve the performance of the toolkit and to allow for simpler binding to GPU processing.

- `Thinc` is an attempt to provide a more type-safe API for a variety of deep learning libraries and frameworks. Although `Thinc` does not directly contribute to the available set of tools, it does provide a cleaner approach for using the underlying tools. Unfortunately, providing a unified access layer to different tools is, in of itself, a troublesome and complex task because not all of the underlying tool concepts can be unified cleanly. As a result it is possible to have "translation" errors when different underlying tools are used.

- `Turi Create` is a development kit designed to make the use of deep learning neural networks and other tools simpler. The focus of the development kit are the `macOS` and `iOS` operating systems. Due to this focus, the available tools within the toolkit are limited to the functionality available within `macOS` and `iOS` devices.

The mentioned tools are relevant to provide an indication of the current trend within the research community, even though deep learning neural networks are not a consideration for this thesis.

11.4 Conclusion

This chapter highlighted the need for reproducible research within CI. Specifically, the environment within which research is performed, the data set file format and then the software used to produce the data for the current study. The use of open-source software and tools for the all aspects of the research process should be encouraged so that changes to the research process itself may be made with the intention to achieve reproducible research outputs.

With focus on current popular tools for EC and SI, the following considerations were highlighted:

- Complex state interactions within software may result in unknown errors and are difficult to locate.

- The use of reflection indicates unsound program behaviour and undermines the value provided by the type system with respects to program validation.

- The use of mutable state within toolkit frameworks and libraries requires extreme programmer discipline to ensure that no unexpected data changes occur.

- Large amounts of user documentation are required to explain correct software usage as well as extension procedures.

- Java as a programming language does very little to discourage incorrect and unsafe programmer actions. Dynamic programming languages discourage even less and allow for trivial runtime code modification.

From the problems listed in the preceding discussion, it is clear that the "status quo" of current tools is insufficient to produce reproducible results. More sophisticated software should instead be sought to address the highlighted problems.

Chapter 12

Monadic Algorithmic Composition

A tension exists between expressive and analytical power; expanding on one will necessarily contract the other. Constraints liberate whilst liberties constrain.

Rúnar Bjarnason

The purpose of abstraction is not to be vague, but to create a new semantic level in which one can be absolutely precise.

Edsgar W. Dijkstra

The need for reproducible and repeatable experiments is well-founded within the scientific method. The sub-field of CI does not have an unblemished reputation in this regard, making the importance of reproducible research highlighted in the preceding chapter even greater. This chapter introduces a redesign of an existing software library for CI which addresses multiple concerns within CI research.

This chapter begins by examining the foundational principles software for CI should maintain within section 12.1. Section 12.2 discusses what a compositional software library entails by delving into functional programming and the associated data structures. From these functional data structures, a new set of data structures are defined within section 12.3 which aim to provide solutions and solve the question of reproducibility of experiments within CI. Lastly, the chapter concludes in section 12.4.

12.1 Foundational Principles

The work within this thesis fostered a new perspective and interpretation for an open-source library for EC and SI. The resulting library is drastically different from the currently available tools that are prevalent within industry and the research community. The library focuses on four main principles related to software for EC and SI algorithms including correctness, type-safety, reproducibility and compositionality.

12.1.1 Correctness

The most important aspect of an algorithm implementation is its correctness. Correctness does, however, need to be specified within the context of algorithmic implementation. The correct implementation of an algorithm defines that the algorithm implementation is the generally accepted implementation (for both industry and research purposes) of an algorithm. The acceptance of the implementation is based on the description and interpretation of the algorithm from the available literature. The open-source nature of the library encourages both peer-review of implementations and the submission of corrections to any mistakes. Correctness should also take priority above any software alterations which may question the correctness of algorithm implementation.

Any performance improvements to the running cost of an algorithm (such as CPU usage, time *etc.*) should only be addressed once the correctness of an algorithm is verified and then reproducibly validated. Algorithm validation should be an automated process and should be randomized using generative testing [39, 126].

As an addition to the correctness of an algorithm implementation, only the use of immutable data is tolerated. Immutable data defines data that cannot change once created. The notion of a variable within a programming language, which implicitly allows for changes to the variable, is instead replaced with the notion of a *value* which cannot change once created. Although the use of immutable data may seem strange and cumbersome, the advantages afforded by immutability far outweigh any potential disadvantage. For example, given a concurrent computer program with many threads, immutable data completely removes the need for data locks between the threads within the program. The different threads may freely use an immutable value without the concern that the value may have changed since the last time it was accessed. Immutable data has also been proven to be performant [215, 216] and is the preferred data model for functional programming languages.

12.1.2 Type Safety

The implementation of CIlib 1.x had an execution structure that relied heavily on the use of reflection and type casting. The `simulator` program necessitated

the reflection usage within the CIlib framework, whilst type casting allowed deep object inheritance hierarchies to correctly expose required functionality. Deep object inheritance hierarchies ultimately produce abstract classes and interfaces (located towards the top of the hierarchy) which are too general. This generality within top level hierarchy members certainly does allow for structures to be treated the same but at the same time, results in a loss of type information for classes in the lower levels of the hierarchy. Common examples of such generality, where the type information loss creates impediments for a program may be found within the standard libraries of both Java [169] and Scala [214]. Consider the Java standard library where data structures are encouraged to implement the `Collection` interface. `Collection` defines a set of operations for all data structures that represent a sequence of elements. In the case of Java, implementations of sequence-like data structures include the implementations for linked lists, sets, maps and queues. Such data structures do not necessarily have similar behaviours. A linked list allows for the retrieval of elements by index, albeit in *O(n)* time, but a set does not. By forcing the `Collection` abstraction, the index-based retrieval function of the linked list is unreachable without first type casting the `Collection` to a more specific type. Programs making use of reflection and type casting can be classified as fragile because the reflection API is based on accesses using strings, which are not checked by a compiler and allow for mistakes that become more difficult for the reader to spot as the strings grow longer. This fragility allows the compilation of a program into a "working program" which will fail when executed because any potential code changes are not observed by the compiler when reflection and casting is used.

The type system available to the programming language should instead be exploited as much as possible. Of course, this consideration is only applicable to programming languages with static type systems. Programming languages with dynamic type systems defer the type verification process to execution time and do not have a compilation process.

Using the type system, truly generic code may be written where only the information necessary for the current operation is considered. The type system can also ensure that invalid program states are not possible to construct. Consider a function which selects an element from a list of elements; the result of the function should be an element within the list. When the list is empty, what should the return value be? It could be that the program terminates with an error condition by throwing an exception or returning the sentinel value of `null`. The problem with this function is that the function simply does not know enough information, nor does the compiler by extension. The solution would be to ensure that the list provided to the function is a `NonEmptyList`, allowing the function to always be able to return a valid result. The `NonEmptyList` data structure is a list structure that is always guaranteed to contain at least one value. Preventing invalid states ensures that errors, such as the mentioned empty list example, are not possible by construction and should rather result

in a compilation failure by the compiler instead of a runtime failure.

Furthermore, using the type system effectively ensures that the need for an intermediate representation (such as the XML based specification of the `simulator` program) is unnecessary.

12.1.3 Reproducibility

Within the research community publications advertise the effectiveness of an approach to solve a given problem. Unfortunately, due to the use of stochastic values within optimization algorithms, the reproduction of results within a publication is generally impossible. The inability to reproduce the presented results impedes the scientific method and prevents the agreement or debunking of the presented publication.

In order to reproduce results, the original program used to produce the published results should be used. It is often the case that a publication provides a description of an algorithm but it is likely that the description is not detailed enough. Due to the enforced length restrictions of journals and conferences the publication authors often opt to forego implementation details in order to allow for the necessary discussion of result analysis. If possible, the publication should provide a reference to an online source for the reader to visit with more detailed algorithm implementation details or the original algorithm implementation itself.

Open-source software built with reproducibility in mind and at the core of the design is highly sought after. Allowing the type system of the implementation language to manage and control the use of stochastic values within the algorithm would provide complete control over the algorithm execution. Through the tracking of stochastic values by the type system, reproducing the results of a publication would require knowledge of both the PRNG initialization seed value and the algorithm definition. The observed execution of the algorithm, when using managed stochastic values, would transform the non-deterministic algorithm execution into an execution that is fully deterministic. Deterministic algorithm execution has the added benefit of allowing for the explicit testing of algorithms without needing to verify the results using "golden" datasets and predefined tolerances to determine equivalent statistical performance.

12.1.4 Composition

Composition is the process of combining existing data structures and functions in order to produce new functions that cater for more specific tasks. The type-safety afforded by the programming language will also ensure that the output value of an operation may be used as the input of the next. Composition facilitates the flow of data by describing the series of transformations to convert a given input into the desired output. By following the transformations of data,

possible errors may be located more easily and will be isolated by the compiler's type-checking process.

Composition also allows for the naming of the resulting transformation. Once a series of transformations are composed into a new transformation, the transformation may be named allowing for reuse at a later time (either by library or user).

Based on the discovery of the above points and the realisation of the deficiencies within the Java version of CIlib, a new implementation began to address these deficiencies and embrace the above fundamental principles.

12.2 A Compositional Library

With focus on the identified requirements described in section 12.1, it was clear that functions would be an important feature of the new implementation of CIlib. Functions implicitly allow for composition when the mathematical notion of a function was considered. Mathematical functions are also *pure*. Pure functions only operate on the provided parameters values in order to produce an output and the same output is produced for the same set of input values. Additionally, pure functions allow for *referential transparency* which defines that a function evaluation may be replaced with its output value for a given set of function input values. The value replacement afforded by referential transparency results in an identical execution of a program, with the only exception being a slightly more efficient program execution because the replacement values have already been calculated.

By considering the benefits of the mathematical function, a programming language viewing functions in the same perspective would be preferred. Functional programming (FP) provides these benefits. Programming languages which support functional programming were therefore considered. Additional knowledge was required because functional programming does not operate in the same manner as object-oriented programming (OOP).

The sections that follow describe the relevance of FP in section 12.2.1 (particularly to stochastic algorithmics), and describe foundational data-structures in section 12.2.2.

12.2.1 Functional Programming

As mentioned in the previous section, FP is a style of programming where computation is modelled as an evaluation of mathematical functions. Functions are *pure* and only operate on the provided function arguments, whilst at the same time eschewing global program state and mutable data. As a result, immutable data is the only consideration when using mathematical functions as the basis of computation.

FP originates from the lambda calculus, a formal system of computation developed by Church [38] to investigate computability, function definition and application, as well as recursion. Later, Church refined the lambda calculus to formalise the simply typed lambda calculus [37] in order to prevent paradoxical uses of the untyped lambda calculus. Over time, additional typed lambda calculi were produced based on the original work of Church. `System F` expanded the typed lambda calculus by allowing universal quantification over types, being independently discovered by logician Girard [99] and computer scientist Reynolds [241]. Universal type quantification allows for the creation of a prepositional function which may be satisfied by every member of a given type. The Curry-style (which associates types with untyped lambda terms) variant of `System F` was proven to have the problem of undecidable type inference by Wells [301].

A restricted view of `System F` that allows for type inference was first described by Hindley [118] and later rediscovered by Milner [196]. The type system was later given a close formal analysis and formal proof by Damas [54] and Damas and Milner [55]. The Hindley-Milner type system, together with its type inference, became the foundational type system for many statically typed FP languages, including Haskell [183] and OCaml [157]. Modern programming languages such as Haskell have, however, since moved onto more expressive logic systems for their type systems.

As a result of programming languages employing logic systems in the form of type systems, an interesting correspondence, known as the Curry–Howard correspondence. The correspondence was discovered by mathematician Haskell Curry and logician William Alvin Howard, whom was the first to make the correspondence explicit [52]. The Curry-Howard correspondence describes a direct relationship between computer programs and mathematical proofs. Exploiting the Curry-Howard correspondence has allowed for computer languages such as Coq [275], in which proofs are seen as programs which can be formalised, checked and executed.

As an example of the relevance of the Curry-Howard correspondence within the research area of EC and SI, Yu and Clack [313] described a software implementation within which the correspondence was proposed as a way to describe search space partition in GP. The method associates indexed sets of genotypes within the GP by their Curry-Howard isomorphic proof (the species).

12.2.2 Functional Structures

Abstraction is a fundamental concept in computer science and mathematics. Abstraction allows for the description of specific events or actions in more general terms. The complexities of writing a computer program are almost never directly visible, yet they exist nonetheless. Abstractions provide a mechanism to broadly discuss ideas and provide a common nomenclature. Furthermore, abstractions also imply a predefined corpus of knowledge is already understood.

For example, discussing a linked-list data structure implies that participants understand the mechanics of the data structure, together with the intricacies of computer memory usage and management.

FP is no different and practitioners should understand the established concepts and nomenclature. In contrast with the programming style of OOP, FP derives much of its terminology from the field of mathematics, particularly the sub-fields of *category theory* and *topology*.

Category theory formalises mathematical structure and concepts through the use of a set of objects (a *category*) and the allowable operations between categories. These operations, known as *arrows* or *morphisms*, formalise transitions between the objects of a category, or *objects* to another category. A category has two basic properties:

1. arrows compose associatively, and
2. each object within a category has an identity arrow (which transitions an object back onto itself).

Category theory has practical applications within programming language theory. The types of data within a programming language (*e.g.*: integers, floating point numbers and user defined structures) form the category of types and the transitions to other types use functions, which are the arrows of the programming language.

The subsequent sections provide a general discussion on general computational contexts, as well as a discussion on the use of optics within FP. The sections thereafter, describe the most fundamental FP abstractions including functors, applicative functors, and monads.

12.2.2.1 Computational Contexts

Each category represents a collection of objects. In turn, these collections of objects may have a specific associated behaviour. Within the subset of categories that are applicable to computation, different behaviours may be present that determine how an object within a category is evaluated. How the evaluation is performed defines the *computational context*. Consider operations that accept a simple recursive cons-list structure. These operations may evaluate the value by de-structuring the cons-list. A trivial example of such an operation is the `length` function. The `length` function traverses the list from head to tail and counts the number of cons-cells within the list. The `length` function may also operate on the provided structure without concerning itself with the kinds of data items contained within the list. Only the structure and shape of the cons-list is important and not the content of the cons cells.

A non-exhaustive but common set of examples of contexts within which values may be calculated include:

- `Maybe` - a context defining the presence or absence of a value. An alternative name for this context is `Option`.
- `Either` - a context maintaining a value or a named error when value is absent
- `List` - a context defining zero or more values
- `NonEmptyList` - a context of one or more values
- `Future` - a value that may be available at a later time
- `Promise` - a value that is guaranteed to exist at a later time

Importantly all contexts can be reasoned about in the same way. That is to say that the definition of a calculation which yields a potentially absent value, or a calculation that will complete at a later time are treated the same. The context for the computation defines how the calculation actually takes occurs.

12.2.2.2 Functor

The functor abstraction represents the mathematical functor, *i.e.*, the mapping between categories in the context of category theory. The mapping process is defined by a structure preserving morphism. This morphism *maps* an object in one category to an object another category. Within the context of computer programming, such morphisms are functions that map one type to another type. For a functor to be sound and to operate as expected, the functor *must* adhere to a set of *laws* which ensure predictable behaviour. Furthermore, these laws also result in a semantic algebra which allows for the independent reasoning of functor behaviour. The laws required for valid functors are:

- **Identity**: the composition of a functor and the identity function, `id`, must produce the original functor

```
fmap id = id
```

- **Composition of morphisms**: if two sequential transitions are performed one after the other using the functions f and g, the result must be the same as a single transition of the composed function of f and g:

```
fmap (f . g) = fmap f . fmap g
```

  the operator `.` is the function composition operator and is pronounced as *after*. *i.e.*, apply function `f` *after* applying function `g` to the input.

Figure 12.1 illustrates the functor laws graphically.

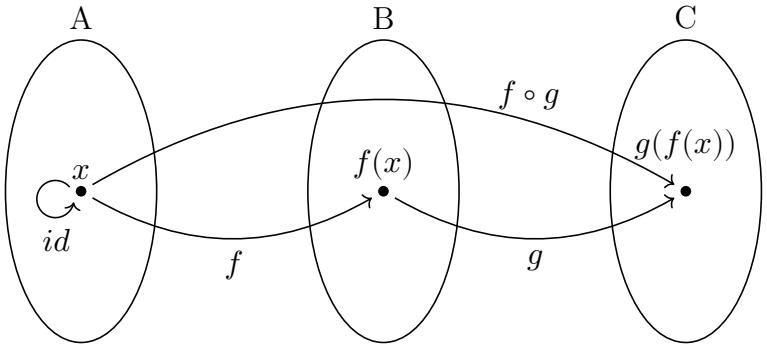


Figure 12.1: Functor morphisms between the categories A, B, and C

12.2.2.3 Applicative Functor

Functors transform a single object in a category to an object in another category. Consider the case where a morphism of more than a single argument is provided to the functor F. An example would be the category of integers and morphisms back onto the category of integers. Self referential functor morphisms are known as *endomorphisms* of the category. If the integer addition function $(+)$ were to be used as the morphism to the category, and applied to an object, the result would no longer be an object within the category of integers, but instead an object within the category of functions on integers. This translation of categories occurs because $+$ is a binary function, requiring two inputs, and not a unary function. Furthermore, such usage would be problematic: there is no longer any mechanism to move from the category of integer functions back to the category of integers because all objects within the category are themselves functions on integers.

Applicative functors [187] are a context aware generalization of functor, which are functors themselves. Applicative functors allow for the context aware representation of functions that accept more than a single input parameter, whereas functions applied to functors require a single input parameter. Importantly, functions within the applicative context are *curried* [187] functions. Composing multiple applicative contexts together allows the applicative functor to ultimately produce a single object within a category as the result of the computation.

The applicative functor extends the functionality of a functor with two additional operations:

- `pure`: an operation to *lift* a value into the context of the applicative functor. An alternative name for `pure` is `point`.

- `apply`: an operation combining an applicative context containing a curried function with other applicative contexts. The combination of contexts requires that the contained objects are suitable for the curried function and are applied in order within the applicative context to pro-

duce the final result. Alternative names for the `apply` operation include `ap` and the infix operator `<*>`.

As with functors, applicative functors require that laws hold in order for the behaviour of the applicative functor to be predictable and well-behaved. The applicative laws extend the laws for functor with:

- Identity laws: The identity function within an applicative context, when applied to an applicative v, must produce v as the result:
  - Left identity:

```
ap(pure(id), v) = v
```

  - Right identity:

```
ap(v, pure(id)) = v
```

- Associativity of morphisms `f`, `g` and `h`:

```
ap(ap(f, g), h) = ap(f, ap(g, h))
```

Figure 12.2 provides an illustration to the process of an applicative functor. The arbitrary applicative context is denoted by the circle shape. A binary function + and the integer values 5 and 3 are within the applicative functor. The result, after the `apply` step composes the applicative contexts, is the value 8 still within the same applicative context. Unlike the functor typeclass, applicative functors allow for the sequencing of computations based on the context of the applicative functor itself. If a given applicative context allows for the parallel computation of results, the parallel nature of the computation itself is not directly observable when executing `apply`. Any execution level effects on the computation runtime are only observed when the final value is evaluated from the applicative functor composition. Using the previous example, the + function within the applicative context may apply either of the parameters in any order because + is an associative function. The application of applicative values also includes partially applying (or binding) a single parameter value. The result of partially applying an applicative functor results in a new applicative functor where the remaining values are expected to produce the final computation value.

12.2.2.4 Monad

In contrast to applicative functors which contain a function within a context, a monad describes a computational context that produces new values within the same context, based on the value of a previous result. The implication is that a monad is able to sequence the order of operations which is the major difference when compared to applicative functors. Moggi [197] published the insight

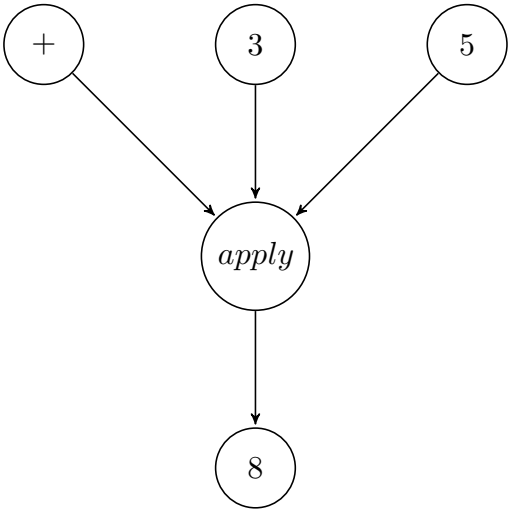


Figure 12.2: Applicative functor application to a binary addition function within the same context

that linked the categorical structure of the same name with computation and functional programming.

Monads allow for the explicit ordering of computation actions and are therefore a more powerful abstraction when compared to applicative functors. Similarly to how all applicative functors are functor instances, all monads are applicative functors as well. The converse, however, does not always hold and the *monad laws* may be tested to prove the validity of a monad instance.

Monad offers an additional operation to the already available operations of both the applicative functor and functor. The additional operation is pronounced *bind* but as with the preceding structures alternative names have also been used to describe the monadic operation. Examples of other naming for the *bind* operation include *flatMap* and the infix symbol `>>=`. Although other names have also been used in different programming languages, these names are not considered because the implementation of monad instances within these languages are invalid; an example of an invalid implementation is the `chain` operation available within JavaScript. Monad literature has also used the names `return` or `unit` for the *pure* operation defined on applicative functors, with `unit` matching the name used in the categorical definition.

Monad instances must observe specific laws in order for the monad instance to be predictable and valid. The first is the identify law for sequencing, whereby an action sequenced with the identity action should produce the original action. The same identity property should hold with the ordering reversed. The remaining law describes the associativity property for monad sequencing.

- Identity laws:
  - Left identity:

Table 12.1: Scala type definitions of three foundational functional structures from category theory

Typeclass	Signature				
Functor	`F[A]`	`=>`	`(A => B)`	`=>`	`F[B]`
Applicative	`F[A]`	`=>`	`F[A => B]`	`=>`	`F[B]`
Monad	`F[A]`	`=>`	`(A => F[B])`	`=>`	`F[B]`

```
pure a >>= f = f a
```

– Right identity:

```
m >>= pure = m
```

- Associativity:

```
(m >>= pure) >>= g = m >>= (\x -> f x >>= g)
```

The sequencing property of monads allows for the description of a computation that may also include side-effects. A side-effect is an additional change to the system executing the computation. Examples of side-effects include the access to the system console to output text, the modification of a file on the filesystem, communication over the network or accessing a database system. Side-effects are only observed when the execution environment executes monadic values. Monadic values need not produce a useful value but may instead exist purely to describe a desired side-effect. Table 12.1 describes the similarities between functor, applicative functor and monad by listing the type signatures of the structures ordered from the least to the most powerful, using Scala [214] syntax.

12.2.2.5 Higher Kinded Types

Programming languages which are able to generically express the functor, applicative and monad types require a type system that is capable of representing higher-kinded types (HKTs). A type system unable to represent HKTs does not preclude the use of all functional programming abstractions, but limits the available options to concrete implementations. For example, for the `List` data structure it is possible to define instances of Functor, Applicative and Monad but to use these instances the `List` structure should always be explicitly referenced. More generic functions and interfaces cannot be expressed using such type systems. The *Typeclassopedia* [312] provides a reference for all the established functional structures, together with the relationships between the structures.

With reference to the previously discussed functor, applicative and monad structures, the following relationships exist between the structures but only when the defined laws hold:

- Functor is the most basic of the three structures.
- All applicative functors are all functors as well, but not all functors are valid applicative functors.
- Monads are all applicative functors as well as functors, but not all applicative functors are monads.

As the functional structures become more specialised (from functor down to monad) more derived operations can be defined based on the primitive operations that the abstractions defines. Conversely, as the structures become more specialised the compositionality of the structures becomes more restricted to the subset of types that can implement these functional structures.

Examples of programming languages which are able to generically express these structures and the associated relationships include Haskell and Scala. The type inference within the Scala language compiler, when using these structures, is limited and often results in compile time type mismatches. The Scala compiler struggles with inference because it considers only local type information for inference. Addressing the type mismatches requires the help of the programmer to manually ascribe types in order to aid the compiler's type checking. Haskell, in contrast, provides better type inference but errors may still occur which require the programmer to ascribe types, albeit far less than what is required when using Scala.

12.2.2.6 Functional References

Values within a deep and nested immutable data structure may be of interest in different usage scenarios. An example may be a specific node within a deep tree data structure. References allow unambiguous pointers into a data structure allowing for the retrieval of the value and possibly the ability to modify the referenced value. The "functional" part of a functional reference refers to the ability of the reference to allow for flexibility and composability expected from pure mathematical functions.

It is already known that immutable values cannot be modified. So, how would a nested value within an immutable value then be updated into a new value? Modification of an immutable value is achieved by copying and updating specific references within the data structure. Creating a copy of an immutable value may at first glance seem an expensive operation which necessitates larger amounts of memory because of data duplication. Creating complete copies of immutable data is wasteful but immutable data can instead make use of a technique known as "structural sharing". Structural sharing reuses as much of

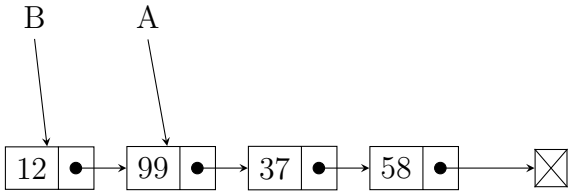


Figure 12.3: Structural sharing of an immutable linked-list

the source data structure from which the copy or update operation is based. The most trivial example of structural sharing is to prepend a new value to the front of an existing immutable linked list. Figure 12.3 illustrates the structural sharing of two linked lists. The list pointed to by the name binding `A` is a three element linked list. `B` is a pointer to a four element linked list, but `B` reuses the data of the linked list pointed to by `A`. Structural sharing affords efficiency of immutable structures by reusing as much of the original data as possible whilst reducing the amount of memory needed to represent the data modification, as also known as "persistent data structures" [216].

Although persistent data structures address the concerns of efficiency, the modification of a nested value may require multiple modifications to intermediate structures within the overall data structure itself. Every level of nesting will require a modification to ensure that the referenced value is updated within the resulting updated persistent structure. These repeated modifications result in a "bubbling up" process until the outer most layer of the data structure is updated.

Functional references have been the research focus of several programming language theory researchers. It was the work by Foster *et al.* [93] on bidirectional programming languages which established the name of "lens" for functional references. A lens is a data structure which exposes two operations: `get` and `set`. These operations are parametrized on a given type, allowing a value to be extracted and to be modified which updates the containing structure. A lens is a first-class value which can be passed around within a program but also allows for a series of specific compositions. Importantly, the compositional nature of the lens allows for "zooming" into more specific values within a data structure. Whilst allowing for this zooming effect the converse action is also considered, whereby modification of a zoomed value will result in all intermediate layers also being updated.

Whilst the lens is the most basic functional reference, other references exist for specific usages and these functional references are collectively referred to as "optics". Optics also allow for functional references to operate on data which may be absent. Even though additional functional references do exist, the discussion of these optics is not necessary in order to establish the functionality which optics provide when working with immutable data structures.

The utility provided by the lens and the optics abstractions when working with immutable structures is preferred when working with nested immutable

data. Optics allowing for a uniform abstraction whilst removing any potential user errors with updating the intermediate layers of nested immutable data.

12.2.2.7 The Classy-optics Pattern

The previous section introduced functional references which allow for the narrowing of data values within a data structure. The classy-optics pattern allows for a constraint definition on a function. The added constraints to a function require that the programming language compiler is able to locate a valid instance of an optic to satisfy the declared function constraint. Convention dictates that classy-optics are prefixed with the word "Has", as if a question is asked about the instances that fulfil the class. An example would be the class `HasVelocity` which requires than an optic instance exists which can retrieve some kind of "velocity" value from a data structure. The requested value type and the data structure itself are type parameters on the classy-optics classes.

If an instance is not available, the result is a compiler error which prevents any invalid usages of the function with the constraint. A function with the `HasVelocity` constraint will not allow values to be used which do not have a velocity data value defined. Such usage is useful to prevent the wrong data structure being passed to the function, only to have the evaluation result in a runtime program error. An example of such usage is the PSO velocity update equation. An initial velocity value is needed in order to obtain the current particle's updated velocity within the update equation. Using an individual from a GA with the velocity update equation would result in a programming error because the constraint ensuring the presence of a velocity value for an individual would not be available, nor defined. Consequently, the explicit definition of classy-optic instances serves to both allow for operations to succeed but also to intentionally disallow others based on the data types passed to these constrained functions.

12.3 Evolutionary Computation Structures

Through the application of the structures introduced in section 12.2.2, it is possible to define well-behaved and law abiding structures for stochastic optimization algorithms. The following sections address the specific aspects of a stochastic optimization algorithm:

- the managed use of randomness as an explicit algorithmic effect,
- composable algorithmic actions,
- algorithm participants (including candidate solution representation and polymorphic candidate solution state), and
- algorithm iteration and execution.

The definitions also allow for the description of an optimization algorithm in a declarative manner, allowing algorithmic pseudo-code that closely mirrors the actual algorithm implementation.

12.3.1 RVar

Within an unmanaged and side-effectful stochastic algorithm, the use of stochastic values within the algorithm are the primary reason for non-reproducible results. The `RVar` data-structure provides the bedrock foundation upon which stochasticity is managed and tracked throughout the definition and execution of an algorithm. Named as the contraction of the term *random variable*, the purpose of the structure is the management of the PRNG state and application of the sampled random values.

Most execution platforms provide a default global PRNG during program execution but it can be argued that this practice is fraught with errors as previously discussed. It is common to find that the default generators often fail statistical randomness tests, including test suites such as the *TestU01* [153] (also known as "Crush") tests and the *DieHard* [184] tests. The test suites attempt to determine if the PRNG algorithms produce values that do not appear to be statistically random. If a pattern is observed within the sampled data, the generator fails the set of tests. The system wide platform random number generator may not provide random samples that are of a sufficient quality for scientific work. Furthermore, the global system generator cannot be controlled by a program and can only be sampled which results in a global system mutation. The CIlib library, therefore, provides a PRNG implementation that is efficient, high quality and sufficient for scientific computation known as the complementary multiply with carry (CMWC) [50] generator.

Converting the sampling process of a PRNG into a pure function results in a function with the shape:

$$\text{PRNG} \rightarrow (\text{PRNG}, \text{A}) \tag{12.1}$$

where PRNG is the state of the random number generator and A is the sampled value from the generator. A sampling function therefore uses a PRNG and produces a tuple result consisting of a PRNG and a generated value represented by the type `A`. The PRNG value within the tuple is a value which contains the modified PRNG state after the completed sampling process. A sampling method may, therefore, use the obtained PRNG value in a subsequent sampling function to obtain the next value within the stream of random values represented by the PRNG. Using the same PRNG value in repeated sampling function invocations will produce identical random values, together with the same PRNG value for the PRNG within the resultant tuple. Consider the following example which uses plain integers and a function, `double` which doubles the integer input: `double`$(2) = 4$. No other input except for the integer 2 will result in the output of 4. However, chaining the doubling function

twice will always result in the expected value of 16, through the evaluation of `double(double(2))`. The exact same process applies to the sampling of a random value because the PRNG is an ordinary value within a program – analogous to using the PRNG as an integer. The exception is that the new PRNG value is not a single value but instead part of a result tuple or pair value.

It is therefore important to carefully consider how subsequent values of the PRNG are used within the subsequent calls to random value sampling functions. The passing of the updated PRNG value into subsequent calls (or the "threading" of the PRNG value) may be a source of errors, especially if the wrong PRNG value is used. The threading should, as a result, be managed by a data structure and not the programmer as the threading process is cumbersome but crucial to ensure correctness.

The `State` monad [293, 294] is a data structure which implements the previously mentioned notion of threading a value though a computation on behalf of the programmer. The resulting monad computation becomes a specialised computation as the state value of the monad becomes fixed. In essence, this is the exact definition of `RVar`: a computational context which uses PRNG values to produce random values whilst managing the complexity of threading updated PRNG values into subsequent `RVar` computations.

All the available monadic operators that may be derived for the monad are implicitly available and valid for use with `RVar` computations. These operations allow for a compositional and declarative use of the PRNG value, whilst guaranteeing the correct threading behaviour of the PRNG.

CIlib provides a collection of functions to produce `RVar` values, which may be composed together in order to produce the values required for a computation. Examples of these combinators include the creation of several random values contained within a `List` structure, the random shuffling of a container of values and the generation of an unending stream of random values. Statistical distributions, such as the Gaussian and Cauchy distributions, may be derived from the combinators available to `RVar`, allowing for computations which require a specific distribution of randomness.

In order to demonstrate the utility afforded by the `RVar` structure, the example that follows should be considered. A triangular distribution function is defined by applying the triangular distribution's cumulative distribution function (CDF) to sampled uniform random variates and is demonstrated within the Scala read-evaluate-proint loop (REPL) within REPL session 12.1. Furthermore, note how the application of the same PRNG value produces identical output values, with the same updated PRNG state at different memory addresses, which is a consequence of the runtime platform.

```
def triangular(a: Double, c: Double, b: Double): RVar[Double] = {
  assert(a <= c)
  assert(c <= b)

  def cdf(x: Double) =
    if (x < a) 0.0
    else if (a <= x && x <= c) ((x-a)*(x-a))/((b-a)*(c-a))
    else if (c < x && x < b) 1.0 - ((b-x)*(b-x))/((b-a)*(b-c))
    else 1.0

  Dist.stdUniform.map(x => {
    val U = cdf(x)
    val F = (c-a)/(b-a)

    if (0 < U && U < F) a + math.sqrt(U*(b-a)*(c-a))
    else b - math.sqrt((1-U)*(b-a)*(b-c))
  })
}

val rng = RNG.init(123456789L)
// rng: RNG = cilib.CMWC@57b9389f
val triangularRVar = triangular(1.0, 2.0, 4.0)
// triangularRVar: RVar[Double] = cilib.RVar$$anon$2@113dcaf8
triangularRVar.run(rng)
// res0: (RNG, Double) = (cilib.CMWC@1201769d, 1.5505102572168221)
triangularRVar.run(rng)
// res1: (RNG, Double) = (cilib.CMWC@5abf6a99, 1.5505102572168221)
```

Scala REPL Session 12.1: Reproducible triangular distribution sampling

12.3.2 Step

Most operators within a stochastic nature-inspired optimization algorithms make use of the following components:

1. The source of random numbers.
2. A function to quantify the quality of a candidate solution.
3. The strategy for the optimization (minimization or maximization) of the problem objective function which serves to guide the optimization algorithm to obtain better quality candidate solutions.

The `Step` data structure builds upon the foundation of `RVar` by considering the additional required properties for an operation within optimization algorithms. A `Step` enriches the `RVar` computational context by supplying an "environment" to the resulting computation. The environment maintains a set of values which enclose the entire `Step` computation. The environment provided to the `Step` contains the optimization problem evaluation function as well as the optimization scheme to follow. Importantly, the enclosing environment represents a *shared* value and this value is available to all composed `Step` computations. A `Step` can, therefore, be regarded as a function from an

environment to a value which may have randomness applied:

$$(\text{Eval}, \text{Opt}) \rightarrow \text{RVar}[\text{A}] \tag{12.2}$$

where `RVar[A]` is an alias to the function defined in equation (12.1).

The environment monad [132] (also termed the reader monad) allows for the composition of functions with a common shared environment to produce a value. Evaluating a reader monad computation produces in a plain value as the result of the computation, whereas equation (12.2) produces a `RVar[A]`. Recall that `RVar[A]` is a state monad specialised to PRNG state values. Therefore, the evaluation of a `Step` computation is a monadic structure which produces another monadic structure.

Unfortunately, it is not possible to define the generic composition of two arbitrary monads [35, 168, 213]. Moggi [197] alluded to the possibility of implementing *monad transformers* and was later implemented by Espinosa [87], Steele [264], and Wadler [293] before being formally proposed by Liang *et al.* [168] within a strongly typed language. A monad transformer applies a specific effect (such as the management of state) on top of another arbitrary monad. By defining `Step` as a monad transformer, the shared environment may be made available to `RVar[A]` computations. Furthermore, the composition problem of arbitrary monads is no longer a concern because the transformer monad allows for the definition of the `bind` operation using the fact that the transformer itself behaves in a particular manner. Monad transformers are not, however, a perfect solution to the observed monad stacking problem [168, 213]. The application order of monad transformers can produce structures which have different behaviour and/or evaluation characteristics. For example, consider the reverse order of the `Step` structure, whereby the shared environment is provided to the computation only after the effect of randomness has been applied. Not only is the resulting program conceptually confusing because all randomness operations necessarily need to have been already evaluated in order to apply the environment, but the optimization scheme would need to direct the algorithm search after the fact. Thankfully, the stacking order for `Step` naturally came about, but transformer ordering should be carefully considered when stacking several monad transformers.

Practically, the `Step` abstraction allows for the definition of algorithmic steps that have clearly defined inputs and output values. Consider the velocity update equation for PSO given in equation (3.1):

$$v_{ij}(t+1) = \omega v_{ij}(t) + c_1 r_{1ij}(y_{ij}(t) - x_{ij}(t)) + c_2 r_{2ij}(\hat{y}_{ij}(t) - x_{ij}(t))$$

The velocity equation creates a new velocity vector for the next iteration of the PSO by computing a linear combination of three different vectors. Implementing the velocity update equation as a `Step` is done in listing 12.1 and closely resembles the original equation.

```
def stdVelocity[S](
    entity: Particle[S, Double],
    social: Position[Double],
    cognitive: Position[Double],
    w: Double,
    c1: Double,
    c2: Double
)(implicit V: HasVelocity[S, Double]): Step[Double, Position[Double]] =
  Step.pointR(for {
    cog <- (cognitive - entity.pos).traverse(x => Dist.stdUniform.map(_ * x))
    soc <- (social - entity.pos).traverse(x => Dist.stdUniform.map(_ * x))
  } yield (w *: V._velocity.get(entity.state)) + (c1 *: cog) + (c2 *: soc))
```

Listing 12.1: `Step` implementation of canonical PSO velocity update equation

12.3.3 Step with State

The `Step` abstraction allows for the definition of algorithmic operations as pure computations. Unfortunately, not all optimization algorithms can be defined as functions from inputs to outputs. Such algorithms compute intermediate values that are used in subsequent iterations of the algorithm itself. The guaranteed convergence particle swarm optimizer (GCPSO) [12] is an example of such an algorithm. A bounding box is maintained around the current global best particle which aids in the refinement of the current best solution. The bounding box either reduces or increases in size based on the current performance of the GCPSO. The bounding box is not valid as an input nor as an output of the GCPSO, but is merely needed for the algorithm to operate correctly.

Runtime algorithm parameters may be included into an algorithm definition by stacking another monad transformer on the `Step` data structure. The `StateT` monad transformer on top of the `Step` structure allows for a predefined algorithm runtime state which is available to the optimization algorithm without altering the behaviour of neither `Step` nor `RVar`. The values within the algorithmic runtime state may be updated as required using the combinators provided by the `StateT` structure.

The GCPSO implementation demonstrating the use of the `StepS` structure, together with the interaction with `Step`, is provided in listing 12.2. The algorithm runtime state is given by the `GCParams` type, with the parameter update process defined on lines 22 to 35.

12.3.4 Position

Locations within the search space of a multi-dimensional optimization problem are referred to as *candidate solutions*. Candidate solutions for an optimization problem may represent one of the following possible cases:

1. **Point**: A candidate solution located within the optimization problem search space, but the quality of the represented solution has not yet been

```
def gcpso[S](w: Double, c1: Double, c2: Double, cognitive: Guide[S, Double])(
   implicit M: HasMemory[S, Double],
            V: HasVelocity[S, Double],
            S: MonadState[StepS[Double, GCParams, ?], GCParams])
: NonEmptyList[Particle[S, Double]] => Particle[S, Double] =>
    StepS[Double, GCParams, Particle[S, Double]] =
 collection =>
   x => {
     val g = Guide.gbest[S]
     for {
       gbest <- StepS.pointS(g(collection, x))
       cog <- StepS.pointS(cognitive(collection, x))
       isBest <- StepS.pointS(Step.pure[Double, Boolean](x.pos eq gbest))
       s <- S.get
       v <- StepS.pointS(
         if (isBest) gcVelocity(x, gbest, w, s)
         else stdVelocity(x, gbest, cog, w, c1, c2))
       p <- StepS.pointS(stdPosition(x, v))
       p2 <- StepS.pointS(evalParticle(p))
       p3 <- StepS.pointS(updateVelocity(p2, v))
       updated <- StepS.pointS(updatePBest(p3))
       failure <- StepS.pointS(
         Step.withCompare[Double, Boolean](
           Comparison.compare(x.pos, updated.pos).andThen(_ eq x.pos)))
       _ <- S.modify(params =>
         if (isBest) {
           params.copy(
             p =
               if (params.successes > params.e_s) 2.0 * params.p
               else if (params.failures > params.e_f) 0.5 * params.p
               else params.p,
             failures = if (failure) params.failures + 1 else 0,
             successes = if (!failure) params.successes + 1 else 0
           )
         } else params)
     } yield updated
 }
```

Listing 12.2: Complete GCPSO algorithm definition

determined.

2. **Solution**: Candidate solutions within the search space of the optimization problem that have evaluated to determine the quality of the represented solution.

The `Position` type is an algebraic data type (ADT). ADTs define a closed set of finite values that inhabit a given type and may define a closed-algebra of operations or functions to operate on ADT values. When considering the `Position` type, re-evaluating a `Solution` value will simply return the provided value because the re-evaluation will not change the result. Similarly, the resultant `Position` from a calculation of other `Position` values will always produce a `Point` value because there is no guarantee that the quality of the resulting `Position` is known. `Position` values will therefore alternate between `Points` and `Solutions` during the operation of the optimization algorithm.

Importantly, due to the well defined set of possible operations, the `Position` will always represent a valid state for a given candidate solution within the search space of the optimization problem. `Position` values allow for vector operations including addition, multiplication and subtraction, amongst others. Additionally, `Position` values maintain the search space boundary information and represent candidate solutions that are within the defined problem search space bounds. An example of `Position` usage and the defined algebra is provided in REPL session 12.2.

12.3.5 Entity

Nature-inspired population based algorithms develop new candidate solutions during the iteration process of the algorithm. Within the literature, each nature-inspired algorithm uses a different metaphor to describe how candidate solutions are transformed to produce new candidate solutions. PSOs use the term *particle* for candidate solutions, whereas GAs refer to candidate solutions as *individuals*. Naturally, other names for candidate solutions also exist, based on the chosen metaphor.

Regardless of the metaphor, a candidate solution for an optimization algorithm consists of more than just a `Position` value. Additional information may be attached to a candidate solution which is required by the optimization algorithm. A generic solution for these different kinds of candidate solution representations allows for a common `Position` value to be combined with an unspecified "state" value. The resulting structure is known as the `Entity` type. `Entity` is a simple product-type [281] that combines a `Position` together with a parametrized state. The definition for `Entity` is given within REPL session 12.3. The value type for the state of the `Entity`, represented by the `S` type parameter, is not defined until the `Entity` is instantiated within a program.

12.3.6 Iteration Scheme

The general structure and shape of an algorithm within EC and SI adheres to the same pattern [221]. Population-based optimization algorithms all traverse a provided collection of `Entity` values to ultimately produce a replacement collection of `Entity` values, which are used as the input for the next algorithm iteration. An `algorithm` is therefore an iterated function from collection of `Entity` to collection of `Entity`. During the execution of the algorithms, each `Entity` is considered separately in order to produce a new `Entity`, which will replace the current `Entity` within the new `Entity` collection. Describing the algorithm structure as a function yields the following signature:

```
NonEmptyList[Entity[S,A]] => Entity[S,A] => Step[A,Entity[S,A]]
```

An algorithm may be iterated in one of two ways:

```
// Basic definition of the spherical benchmark problem
val spherical =
  Eval.unconstrained[NonEmptyList,Double](_.map(x => x * x).suml).eval

// The Position is the core data structure to define search space
// locations.
val domain = Interval(-100.0, 100.0) ^ 3

val p1 = Position(NonEmptyList(1.0,2.0,3.0), domain)
val p2 = Position(NonEmptyList(6.0,7.0,8.0), domain)

// Evaluating the quality of a solution promotes the representation to
// a Solution. An RNG is required to cater for potential usage of
// randomness
val p1Solution = Position.eval(spherical, p1).eval(rng)

// Point + Solution = Point
p1 + p1Solution
// res0: Position[Double] = Point(
//   NonEmpty[2.0,4.0,6.0],
//   NonEmpty[[-100.0, 100.0],[-100.0, 100.0],[-100.0, 100.0]]
// )

// Scalar * Solution = Point
3 *: p1Solution
// res1: Position[Double] = Point(
//   NonEmpty[3.0,6.0,9.0],
//   NonEmpty[[-100.0, 100.0],[-100.0, 100.0],[-100.0, 100.0]]
// )

// Solution - Point = Point
p1Solution - p1
// res2: Position[Double] = Point(
//   NonEmpty[0.0,0.0,0.0],
//   NonEmpty[[-100.0, 100.0],[-100.0, 100.0],[-100.0, 100.0]]
// )

// Solution + Solution = Point
p1Solution + p1Solution
// res3: Position[Double] = Point(
//   NonEmpty[2.0,4.0,6.0],
//   NonEmpty[[-100.0, 100.0],[-100.0, 100.0],[-100.0, 100.0]]
// )

// Negate Solution = Point
-p1Solution
// res4: Position[Double] = Point(
//   NonEmpty[-1.0,-2.0,-3.0],
//   NonEmpty[[-100.0, 100.0],[-100.0, 100.0],[-100.0, 100.0]]
// )
```

Scala REPL Session 12.2: Example of `Position` usage and algebra

1. Synchronous iteration, or
2. Asynchronous iteration

The synchronous algorithm iteration scheme is also known as the generational iteration scheme. Synchronous iteration produces new `Entity` values by

```scala
final case class Entity[S,A](state: S, position: Position[A])

// Create a type alias to Individual, which is an Entity with a "Unit" state.
// `Unit` is a type with a single valid value: ()
type Individual[A] = Entity[Unit, A]

// Create a type alias to Particle, which contains a memory that maintains
// the best previous position and the current velocity of the particle.
// The memory items are defined within the `Mem` type
final case class Mem[A](best: Position[A], velocity: Position[A])
type Particle[A] = Entity[Mem[A], A]

// Examples of creating `Entity`s
val pointInSearchSpace = NonEmptyList(0.0, 1.0, 2.0)
// pointInSearchSpace: NonEmptyList[Double] = NonEmpty[0.0,1.0,2.0]
val searchSpaceDomain  = Interval(0.0, 3.0) ^ 3
// searchSpaceDomain: NonEmptyList[Interval[Double]] = NonEmpty[[0.0, 3.0],[0.0,
↪  3.0],[0.0, 3.0]]
val position = Position(pointInSearchSpace, searchSpaceDomain)
// position: Position[Double] = Point(
//   NonEmpty[0.0,1.0,2.0],
//   NonEmpty[[0.0, 3.0],[0.0, 3.0],[0.0, 3.0]]
// )

val anIndividual: Individual[Double] = Entity((), position)
// anIndividual: Individual[Double] = Entity(
//   (),
//   Point(NonEmpty[0.0,1.0,2.0], NonEmpty[[0.0, 3.0],[0.0, 3.0],[0.0, 3.0]])
// )
val aParticle: Particle[Double] = Entity(Mem(position, position.zeroed), position)
// aParticle: Particle[Double] = Entity(
//   Mem(
//     Point(NonEmpty[0.0,1.0,2.0], NonEmpty[[0.0, 3.0],[0.0, 3.0],[0.0, 3.0]]),
//     Point(NonEmpty[0.0,0.0,0.0], NonEmpty[[0.0, 3.0],[0.0, 3.0],[0.0, 3.0]])
//   ),
//   Point(NonEmpty[0.0,1.0,2.0], NonEmpty[[0.0, 3.0],[0.0, 3.0],[0.0, 3.0]])
// )
```

Scala REPL Session 12.3: Generic `Entity` definition and usage

only considering the current collection of `Entity` values. The next collection of `Entity` values is produced by applying the algorithm function with the current `Entity` collection to every `Entity` within the current collection. The result of this process is a subsequent `Entity` collection, which replaces the current `Entity` collection in the following algorithm iteration. Alternatively, an algorithm may iterate using an asynchronous or steady-state iteration scheme. With the asynchronous iteration scheme, the `Entity` collection input to the algorithm function consists of the current `Entity` collection, together with the partially formed next `Entity` collection. The asynchronous `Entity` collection is constructed by substituting replacement `Entity` values with those that have already been constructed for the new collection. This results in an `Entity` collection that contains zero replacement `Entity` values (requiring that replacements for all `Entity` values are still to be produced) to the last scenario, where all but one replacement `Entity` value (for the last `Entity`) is required. An

example of `Entity` replacement during the asynchronous iteration scheme is provided in table 12.2.

Table 12.2: Creation of `Entity` collection during asynchronous iteration. The $\oplus$ operator combines the current and next `Entity` collections to create a collection of n `Entity` values.

Entity	Current collection		Next collection		Result
e_1	$\{e_1, e_2, \dots, e_n\}$	$\oplus$	$\{\}$	$\rightarrow$	e_1^+
e_2	$\{e_2, e_3, \dots, e_n\}$	$\oplus$	$\{e_1^+\}$	$\rightarrow$	e_2^+
e_3	$\{e_3, e_4, \dots, e_n\}$	$\oplus$	$\{e_1^+, e_2^+\}$	$\rightarrow$	e_3^+
e_4	$\{e_4, e_5, \dots, e_n\}$	$\oplus$	$\{e_1^+, e_2^+, e_3^+\}$	$\rightarrow$	e_4^+
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
e_{n-1}	$\{e_{n-1}, e_n\}$	$\oplus$	$\{e_1^+, e_2^+, \dots, e_{n-2}^+\}$	$\rightarrow$	e_{n-1}^+
e_n	$\{e_n\}$	$\oplus$	$\{e_1^+, e_2^+, \dots, e_{n-1}^+\}$	$\rightarrow$	e_n^+

From the description of the iteration schemes for an algorithm, it should be noted that *only* the synchronous strategy is able to benefit from parallel execution. The asynchronous strategy requires a `Entity` collection which is formed by the merger of the current and next `Entity` collections. A race condition would exist when using the asynchronous strategy together with parallelism. The race condition results in a question as to which `Entity` collection is the correctly updated `Entity` collection which should be considered for the current `Entity` within the algorithm function defined in the previous section. Iteration scheme functions are able to transform the algorithm function shape into functions that produce a new `Entity` collection from the provided `Entity` collection. REPL session 12.4 demonstrates the creation of both the synchronous and the asynchronous GBest PSO algorithms.

12.3.7 Runner

Once the algorithm, iteration scheme, `Entity` collection and evaluation environment are defined, a single iteration of the algorithm may be completed. The result of the algorithm iteration is a new `Entity` collection, which may be fed into a following algorithm iteration as the input `Entity` collection. The iterated algorithm function evaluation continues until some stopping condition is reached.

The `Runner` abstraction defines an execution process which allows for the repeated evaluation of an algorithm by chaining the output `Entity` collection back as input to the algorithm once again. Mathematically, when considering the declarative behaviour CIlib provides, the `Runner` abstraction evaluates an iterated function, which may or may not be a fixed-point evaluation, until the defined stopping condition. A fixed point evaluation is only possible when

```
val bounds = Interval(-5.12, 5.12) ^ 5
val spherical =
  (xs: NonEmptyList[Double]) => xs.foldLeft(0.0)((a,c) => a + c * c)

val env =
  Environment(
    cmp = Comparison.dominance(Min),
    eval = Eval.unconstrained(spherical).eval
  )

/** Define a normal GBest PSO */
val cognitive = Guide.pbest[Mem[Double], Double]
val social    = Guide.gbest[Mem[Double]]
val gbestPSO  = gbest(0.729844, 1.496180, 1.496180, cognitive, social)

/** Create the synchronous iteration version of the gbestPSO algorithm */
val syncGBestPSO = Iteration.sync(gbestPSO)
// syncGBestPSO: Kleisli[Step[Double, $0$], NonEmptyList[Particle[Mem[Double],
↪  Double]], NonEmptyList[Particle[Mem[Double], Double]]] = Kleisli(
//   cilib.Iteration$$$Lambda$1585/897478931@64d1f549
// )

val asyncGBestPSO = Iteration.async(gbestPSO)
// asyncGBestPSO: Kleisli[Step[Double, $2$], NonEmptyList[Particle[Mem[Double],
↪  Double]], NonEmptyList[Particle[Mem[Double], Double]]] = Kleisli(
//   cilib.Iteration$$$Lambda$1586/1721002441@6addfa22
// )
```

Scala REPL Session 12.4: Synchronous and asynchrouns iteration of the same PSO algorithm

the combination of the optimization algorithm, initial `Entity` collection and PRNG seed value result in convergence for a given optimization problem.

`Runner` provides the user with a simplified execution entry point within CIlib. However, if a more complex or customised execution process is required, it may freely be defined using the base abstractions within CIlib. REPL session 12.5 demonstrates a sample `Runner` usage by defining and executing a GBest PSO.

12.4 Conclusion

This chapter proposed an alternative to the popular use of OOP for CI software. The importance and urgency to provide CI software that maintains correctness and reproducibility is fundamental to ensuring improvements, especially as optimisation problems and algorithms become more complex. This chapter postulated that the complexity of more advanced algorithms and problems could be controlled through the use of functional programming. Functional programming allows the user to cleanly define the data flows and transformations which constitute a program. Such transformations have been abstracted to form foundational data types which improve the understanding of programs

```
// Bounds of the problem search space
val bounds = Interval(-5.12, 5.12) ^ 10

// Execution environment definition
val env =
  Environment(
    cmp = Comparison.dominance(Min),
    eval =
      Eval.unconstrained((p: NonEmptyList[Double]) =>
        p.map(x => x * x).suml
      ).eval
  )

// Define a "normal" GBest PSO
val cognitive = Guide.pbest[Mem[Double], Double]
val social = Guide.gbest[Mem[Double]]
val gbestPSO = gbest(0.729844, 1.496180, 1.496180, cognitive, social)

// Initial particle swarm - 20 particles within defined bounds
val swarm =
  Position.createCollection(
    PSO.createParticle(x => Entity(Mem(x, x.zeroed), x)))(bounds, 20)

// Synchronous PSO
val iter = Iteration.sync(gbestPSO)

// Static problem landscape (1000 repeated identical landscapes)
val problemStream =
  Runner.staticProblem("spherical", env.eval, RNG.init(123L)).take(1000)

val t = Runner.foldStep(
  env,
  RNG.fromTime,
  swarm,
  Algorithm("gbestPSO", iter),
  problemStream,
  (x: NonEmptyList[Particle[Mem[Double], Double]]) => RVar.point(x))

val executionResult = t.runLast.unsafePerformSync

executionResult match {
  case None => "Execution failed"
  case Some(progress) =>
    progress.value.head.pos.toString
}
// res0: String = "Solution(NonEmpty[-2.785530299488214E-25,9.517101498797723E-
↪  24,4.904807598345465 3E-23,-8.941157286234897E-23,6.6482067721303 86E-23,-
↪  1.3727734757414276E-22,4.388028132394903E-23,-2.67988277712035 7E-
↪  23,2.037755027469882 5E-23,-2.5814879300891756E-23],NonEmpty[[-5.12,
↪  5.12],[-5.12, 5.12],[-5.12, 5.12],[-5.12, 5.12],[-5.12, 5.12],[-5.12,
↪  5.12],[-5.12, 5.12],[-5.12, 5.12],[-5.12, 5.12],[-5.12,
↪  5.12]],Single(Feasible(3.748104022511585E-44),List()))"
```

Scala REPL Session 12.5: GBest PSO definition executed by the defined `Runner` abstraction

by enforcing type constraints through typeclasses. Through the use of these foundational data types within functional programming it is feasible to specify pure computations that may be composed and passed around to other computations as simple values. From the description of pure functional programming, a set of data structures were defined that not only addressed the raised concerns for CI software, but provided the benefits afforded through functional programming to CI.

The result of this chapter was a re-imagining of the CIlib software library as a monadic, composable software library. CIlib allows the complexities introduced and considered in previous chapters of this thesis to be defined, managed and allows for perfectly reproducible algorithmic experiments. Through the use of CIlib and the development of the library, the experimental work within this thesis would not have been possible nor explainable. Finally, the version of the software used within this thesis may possibly no longer match what is available within the project repository online. Even so, the fundamental design discussed within this chapter will largely remain the same.

Chapter 13

Conclusion

This chapter begins with a summary of the findings and contributions of the thesis in section 13.1, followed by a discussion of potential future work in section 13.2.

13.1 Summary of Conclusions

The primary objectives of this thesis were to analyse the influence of optimisation problem complexity on the performance of optimisation algorithms, whilst considering the four different categories of DCOPs, namely SOSC, SODC, DOSC and DODC.

The first objective of this thesis was to provide a summary of the overall optimisation process as it pertains to optimisation problems of increasing complexity. Chapter 2 began the discussion on the optimisation process by focusing on the different optimisation problem types. Due to the established inter-dependency between the optimisation problem and problem search space constraints, an extensive overview of constraint handling approaches was also discussed. The inter-dependency between the optimisation problem and the problem constraints cannot simply be separated because the underlying optimisation problem is transformed by the problem constraints. Furthermore, the presence of optimisation problem constraints has a direct influence on the operation and execution of the optimisation algorithm tasked to find solutions within such problem search spaces. Following on from the discussion of optimisation problems, a subset of EC and SI optimisation algorithms were discussed within chapter 3. The different classes of optimisation algorithms were enumerated, building on the complexity of algorithms and ended with the inclusion of constraint handling approaches which direct the algorithmic search away from infeasible search space regions. As a result, the inherit complexity of the optimisation process was established and provided a foundation for the main work within this thesis.

The second objective within this thesis was to investigate the complexity of

optimisation problems that include dynamic and constrained problem search spaces. Chapter 5 investigated the complexity associated with the optimisation problem by describing the problem landscape using a comprehensive classification system. Based on the findings about current optimisation problems, with emphasis on DOPs, several existing benchmark problem instances and problem generators were investigated to understand how much of the possible problem complexity they expose to the optimisation algorithm. Building on the investigation, optimisation problems that include constraints were examined and found to be rather simplistic and limited, being unable to express the demanding complexity of convoluted DODC problems. As a result, a new benchmark function generator was developed and is proposed for research into DCOPs. Even though the resulting benchmark instances are complex, they are built from well-understood component landscapes which compose to produce the final problem instance. The strengths of the generated problem landscapes was then tested using FLA which showed that the resulting benchmark function generator can produce highly dynamic and constrained problem instances. The results of the landscape analysis indicated the proposed benchmark generator could produce the most complex of problem landscapes, at any level of dimensionality.

The third objective was to determine how to effectively measure the performance of algorithms operating on DCOP. Chapter 6 proposed a new performance benchmark which is not based on scalar values but is rather vector based, defining the *performance profile* of the optimisation algorithm which may be compared to a target (or estimate) performance. From the currently available performance measures, it was concluded that the existing measures are only able to provide a partial understanding of algorithm performance with the necessary problems raised and highlighted. The proposed performance measure, P_{RED}, provides an unbiased view of the algorithm performance based on the distance to the target solution over a number of algorithm iterations. P_{RED} was found to be flexible enough to also describe the performance of optimisation algorithms within SOPs and to also consider performances of algorithms influenced by problem constraints, such as DCOPs.

Building on the work established within chapters 5 and 6, a comprehensive set of DCOP benchmark problems were evaluated to investigate the complexity of the optimisation problem within the optimisation process. The empirical work within chapter 10 evaluated a set of optimisation algorithms on the comprehensive set of benchmark problem instances to address the fourth objective of this thesis. The objective considered if current optimisation algorithms were able to solve DCOPs and how different constraint handling approaches would impact algorithm performance. Within this chapter, the empirical results were evaluated from both the perspectives of the objective and constraint spaces of the optimisation problem. These perspectives were considered during the maintenance and introduction of diversity, algorithm solution accuracy and the recovery of algorithms once the problem landscape experiences a change.

It was found that the data suggested that the choice of constraint handling approach for the algorithm has a significant influence in the overall algorithm performance, and that algorithms could manage to provide solutions to the comprehensive set of benchmark problems. Furthermore, it was found that current algorithms are not effective in all usages, particularly when contrasting SOP and DOP problem spaces. Once constraints are added to the problem definition, more algorithms start to become ineffective with both the algorithm and constraint handling approach becoming equally important considerations. This finding was only possible due to a by-product of this thesis; namely a CI software library that could control and maintain the randomness within the algorithm executions, thereby providing for fair analysis and perfect experiment reproduction. Moreover, DCOP problem landscapes had better solutions provided by algorithms that provided self-adaption characteristics.

The fifth and final objective of the thesis resulted in a consistent approach to CI algorithm implementations that allows for the perfect replication of experimental results. Chapter 11 investigated the need for reproducible research, highlighting the preference for open-source software to allow for complete transparency between researchers. Furthermore, this chapter proposed and substantiated the requirements for data formats, experimentation environments and how reproducible results provide a benefit to research in general. Lastly, chapter 12 described the implementation of a software library, built with the findings of chapter 11 in mind. The new software library was compared to multiple existing alternatives and the problems with each was identified. The final result was a software library that is truly compositional, using functional programming, in order to aid researchers in managing the increasing complexity of problem, algorithm and constraint handling implementations. This same software library, CIlib, was the vehicle within which the empirical work for this thesis was conducted.

13.2 Future Work

A number of avenues for future research are possible based off of the observations and conclusions from this thesis.

With the development of the CMPB benchmark problem generator it may be fruitful to consider how the benchmark problem generator adjusts when different types of peaks are used for either the objective or the constraint spaces. The addition or adjustment of the peak shape will further alter the composed problem landscapes and may expose alternative considerations for the algorithm. Additionally, another possible avenue for research is to use the perceived landscape characteristics of the generated benchmark problem to aid the optimisation algorithm to adapt in order to guide the search through the landscape.

Vector based performance measures have shown a clear benefit from this

thesis. The diversity measurements, whilst informative, provide a scalar value that is difficult to consider without context. It is unclear if the observed diversity value is desired and the choice of algorithm complicates the meaning further. For example, larger diversity values may be relevant for algorithm A but the same diversity value for algorithm B may be completely insufficient or simply not required. As a result, a meaningful vector based diversity measure may provide more value than the currently available diversity measures.

From the observations about the performance of algorithms on the provided benchmark problems, new algorithms might be developed that consider the dynamism of DCOP benchmarks a primary focus. Static optimisation algorithms are merely a special case of this form of algorithm. Good results might be obtained from an algorithm designed explicitly for DCOPs, being simplified for both unconstrained dynamic and static problem landscapes.

Possibly the most important finding is the evidence of a relationship existing between the optimisation algorithm and the constraint handling approach that is far more relevant and important to consider within DCOPs. It is hypothesised that the relationship will become increasingly important as the dimensionality of the optimisation problem increases. As a result, the combination of both optimisation algorithm and constraint handling approach will have a remarked impact on the ability of the optimisation algorithm to provide feasible solutions to a DCOP problem landscape.

Further research into the use of reproducible experimentation to avoid and correct the errors present within CI results should be considered. If the production of results were to be standardised to a procedure that was fully transparent and allowed for almost effortless result reproduction, a number of unanswered questions might be addressed when they arise from publication reviewers and readers. The ultimate benefit of this approach to research queries will ensure a high standard and will allow for simpler refutations of claims, especially as the current research trend is to focus on positive results only.

Bibliography

[1] C. Aggarwal, A. Hinneburg, and D. Keim. "On the Surprising Behavior of Distance Metrics in High Dimensional Space". In: *Proceedings of the International Conference on Database Theory*. Ed. by J. Van den Bussche and V. Vianu. Vol. 1973. Lecture Notes in Computer Science. Oct. 2001. DOI: 10.1007/3-540-44503-X_27.

[2] M. Ameca-Alducin, E. Mezura-Montes, and N. Cruz-Ramirez. "Differential Evolution with Combined Variants For Dynamic Constrained Optimization". In: *Proceedings of the IEEE Congress on Evolutionary Computation*. 2014, pp. 975–982. DOI: 10.1109/CEC.2014.6900629.

[3] P. J. Angeline. "Tracking Extrema in Dynamic Environments". In: *Proceedings of the International Conference on Evolutionary Programming*. Springer. 1997, pp. 335–345. DOI: 10.1007/BFb0014823.

[4] *Apache Arrow*. Oct. 2016. URL: https://arrow.apache.org/ (visited on 04/12/2019).

[5] *Apache Parquet*. Mar. 2013. URL: https://parquet.apache.org/ (visited on 01/21/2019).

[6] T. Bäck. "On the Behavior of Evolutionary Algorithms In Dynamic Environments". In: *Proceedings of the IEEE International Conference on Evolutionary Computation*. May 1998, pp. 446–451. DOI: 10.1109/ICEC.1998.699839.

[7] T. Bäck and S. Khuri. "An Evolutionary Heuristic for the Maximum Independent Set Problem". In: *Proceedings of the IEEE Conference on Evolutionary Computation*. Vol. 2. 1994, pp. 531–535. DOI: 10.1109/ICEC.1994.350004.

[8] J. E. Baker. "Reducing Bias and Inefficiency in the Selection Algorithm". In: *Proceedings of the International Conference on Genetic Algorithms on Genetic Algorithms and Their Application*. 1987, pp. 14–21.

[9] H. J. C. Barbosa. "A Coevolutionary Genetic Algorithm for Constrained Optimization". In: *Proceedings of the IEEE Congress on Evolutionary Computation*. Vol. 3. 1999, pp. 1605–1611. DOI: 10.1109/CEC.1999.785466.

[10] M. S. Bazaraa, H. D. Sherali, and C. M. Shetty. *Nonlinear Programming: Theory and Algorithms*. 3rd ed. John Wiley & Sons, Inc. 872 pp. ISBN: 978-0-471-48600-8.

[11] J. L. Bentley. "Multidimensional Binary Search Trees Used for Associative Searching". In: *Communications of the ACM* 18.9 (Sept. 1975), pp. 509–517. DOI: 10.1145/361002.361007.

[12] F. van den Bergh and A. P. Engelbrecht. "A New Locally Convergent Particle Swarm Optimizer". In: *Proceedings of the IEEE International Conference On Systems, Man, and Cybernetics*. Vol. 7. 2002, pp. 6–9. DOI: 10.1109/ICSMC.2002.1176018.

[13] D. P. Bertsekas. *Constrained Optimization and Lagrange Multiplier Methods*. English. 1982. ISBN: 978-0-12-093480-5. DOI: 10.1016/C2013-0-10366-2.

[14] K. S. Beyer *et al.* "When Is "Nearest Neighbor" Meaningful?" In: *Proceedings of the International Conference on Database Theory*. 1999, pp. 217–235. DOI: 10.5555/645503.656271.

[15] J. Bezanson *et al.* "Julia: a Fresh Approach to Numerical Computing". In: *SIAM Review* 59.1 (2017), pp. 65–98. DOI: 10.1137/141000671.

[16] G. Bilchev and I. C. Parmee. "The Ant Colony Metaphor for Searching Continuous Design Spaces". In: *Evolutionary Computing* (1995). Ed. by T. C. Fogarty, pp. 25–39. DOI: 10.1007/3-540-60469-3_22.

[17] T. M. Blackwell and J. Branke. "Multi-Swarm Optimization in Dynamic Environments". In: *Proceedings of the Workshops on Applications of Evolutionary Computation*. Springer. 2004, pp. 489–500. DOI: 10.1007/978-3-540-24653-4_50.

[18] T. M. Blackwell and J. Branke. "Multiswarms, Exclusion, and Anti-Convergence In Dynamic Environments". In: *IEEE Transactions on Evolutionary Computation* 10.4 (), pp. 459–472. DOI: 10.1109/TEVC.2005.857074.

[19] T. Blackwell and P. Bentley. "Don't push me! Collision-avoiding swarms". In: *Proceedings of the IEEE Congress on Evolutionary Computation*. Vol. 2. IEEE. Feb. 2002, pp. 1691–1696. DOI: 10.1109/CEC.2002.1004497.

[20] T. Blackwell, J. Branke, and X. Li. "Particle Swarms for Dynamic Optimization Problems". In: *Swarm Intelligence: Introduction and Applications*. Ed. by C. Blum and D. Merkle. Springer Berlin Heidelberg, 2008, pp. 193–217. ISBN: 978-3-540-74089-6. DOI: 10.1007/978-3-540-74089-6_6.

[21] J. Bossek. "ECR 2.0: A Modular Framework for Evolutionary Computation in R". In: *Proceedings of the Genetic and Evolutionary Computation Conference Companion*. 2017, pp. 1187–1193. DOI: 10.1145/3067695.3082470. URL: http://doi.acm.org/10.1145/3067695.3082470.

[22] A. Boumaza. "Learning Environment Dynamics from Self-Adaptation: A Preliminary Investigation". In: *Proceedings of the Workshop on Genetic and Evolutionary Computation*. 2005, pp. 48–54. DOI: 10.1145/1102256.1102265.

[23] J. Branke. *Evolutionary Optimization in Dynamic Environments*. Norwell, MA, USA: Kluwer Academic Publishers, 2001. ISBN: 0792376315. DOI: 10.1007/978-1-4615-0911-0.

[24] J. Branke. "Memory Enhanced Evolutionary Algorithms for Changing Optimization Problems". In: *Proceedings of the IEEE Congress on Evolutionary Computation*. Vol. 3. 1999. DOI: 10.1109/CEC.1999.785502.

[25] J. Branke. "Evolutionary Approaches to Dynamic Optimization Problems - Updated Survey". In: *GECCO Workshop on Evolutionary Algorithms for Dynamic Optimization Problems* (Jan. 2001).

[26] J. Branke, E. Salihoğlu, and Ş. Uyar. "Towards an Analysis of Dynamic Environments". In: *Proceedings of the Conference on Genetic and Evolutionary Computation*. 2005, pp. 1433–1440. DOI: 10.1145/1068009.1068237.

[27] J. Branke and H. Schmeck. "Designing Evolutionary Algorithms for Dynamic Optimization Problems". In: *Advances in Evolutionary Computing: Theory and Applications*. Berlin, Heidelberg: Springer-Verlag, 2003, pp. 239–262. ISBN: 3540433309. DOI: 10.5555/903758.903768.

[28] H. Bremermann. *Optimization through Evolution and Recombination*. Ed. by M. Yovtis, G. Jacobi, and G. Goldstein. Self-Organization Systems. Spartan Books, 1962, pp. 93–106.

[29] L. T. Bui, H. A. Abbass, and J. Branke. "Multiobjective Optimization for Dynamic Environments". In: *Proceedings of the IEEE Congress on Evolutionary Computation*. Vol. 3. 2005, pp. 2349–2356. DOI: 10.1109/CEC.2005.1554987.

[30] B. Calvo and G. Santafe. "scmamp: Statistical Comparison of Multiple Algorithms in Multiple Problems". In: *The R Journal* Accepted for publication (2015).

[31] E. Cantú-Paz. "Migration Policies and Takeover Times in Parallel Genetic Algorithms". In: *Proceedings of the Genetic and Evolutionary Computation Conference*. Ed. by W. Banzhaf *et al.* 1999, p. 775.

[32] A. Carlisle and G. Dozier. "Adapting Particle Swarm Optimization to Dynamic Environments". In: *Proceedings of the International Conference on Artificial Intelligence*. 2000, pp. 429–434.

[33] S. E. Carlson and R. Shonkwiler. "Annealing a Genetic Algorithm Over Constraints". In: *Proceedings of the IEEE International Conference on Systems, Man, and Cybernetics*. Vol. 4. 1998, pp. 3931–3936. DOI: 10.1109/ICSMC.1998.726702.

[34] S. Cheng and Y. Shi. "Normalized Population Diversity in Particle Swarm Optimization". In: *Proceedings of the International Conference in Swarm Intelligence*. Ed. by Y. Tan *et al.* 2011, pp. 38–45. DOI: 10.1007/978-3-642-21515-5_5.

[35] P. Chiusano and R. Bjarnason. *Functional Programming in Scala*. 1st. USA: Manning Publications Co., 2014. ISBN: 1617290653.

[36] C. C.-J. Chung and R. Reynolds. "A Testbed for Solving Optimization Problems Using Cultural Algorithms". In: *Proceedings of the Conference on Evolutionary Programming*. Ed. by P. J. A. Lawrence J. Fogel and T. Bäck. Mar. 1999.

[37] A. Church. "A Formulation of the Simple Theory of Types". In: *The Journal of Symbolic Logic* 5.2 (1940), pp. 56–68. URL: http://www.jstor.org/stable/2266170.

[38] A. Church. "A Set of Postulates for the Foundation of Logic". In: *Annals of Mathematics* 33.2 (1932), pp. 346–366. URL: http://www.jstor.org/stable/1968337.

[39] K. Claessen and J. Hughes. "QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs". In: *ACM SIGPLAN Notices* 35.9 (Sept. 2000), pp. 268–279. DOI: 10.1145/357766.351266.

[40] C. W. Cleghorn and A. P. Engelbrecht. "Particle Swarm Variants: Standardized Convergence Analysis". In: *Swarm Intelligence* 9 (2 2015), pp. 177–203.

[41] C. W. Cleghorn. "Particle Swarm Optimization: Empirical and Theoretical Stability Analysis". PhD thesis. Pretoria, Gauteng, South Africa: University of Pretoria, 2017.

[42] T. Cloete, A. Engelbrecht, and G. Pamparà. "CIlib: a Collaborative Framework for Computational Intelligence Algorithms - Part II". In: *Proceedings of the IEEE International Joint Conference on Neural Networks*. June 2008, pp. 1764–1773. DOI: 10.1109/IJCNN.2008.4634037.

[43] H. G. Cobb. *An Investigation into the Use of Hypermutation As An Adaptive Operator in Genetic Algorithms Having Continuous, Time Dependent Nonstationary Environments*. Tech. rep. NRL-MR-6760. Washington DC: Naval Research Lab, 1990.

[44] H. G. Cobb and J. J. Grefenstette. "Genetic Algorithms for Tracking Changing Environments". In: *Proceedings of the International Conference on Genetic Algorithms*. 1993, pp. 523–530. DOI: 10.5555/645513.657576.

[45] C. A. C. Coello. "Self-Adaptive Penalties for GA-Based Optimization". In: *Proceedings of the IEEE Congress on Evolutionary Computation*. Vol. 1. 1999, pp. 573–580. DOI: 10.1109/CEC.1999.781984.

[46] C. A. C. Coello. *A Survey of Constraint Handling Techniques used with Evolutionary Algorithms*. Tech. rep. Laboratorio Nacional de Informática Avanzada, 1999.

[47] C. A. C. Coello. *Theoretical and Numerical Constraint-Handling Techniques used with Evolutionary Algorithms: A Survey of the State of the Art*. 2002.

[48] D. W. Coit and A. E. Smith. "Penalty Guided Genetic Search for Reliability Design Optimization". In: *Computers & Industrial Engineering* 30.4 (1996), pp. 895–904. DOI: 10.1016/0360-8352(96)00040-X.

[49] M. A. Coletti, E. O. Scott, and J. K. Bassett. "Library for Evolutionary Algorithms in Python (LEAP)". In: *Proceedings of the Genetic and Evolutionary Computation Conference Companion*. 2020, pp. 1571–1579. DOI: 10.1145/3377929.3398147.

[50] R. Couture and P. L'Ecuyer. "Distribution Properties of Multiply-With-Carry Random Number Generators". In: *Mathematics of Computation* 66.218 (1997), pp. 591–607. DOI: 10.1090/S0025-5718-97-00827-2.

[51] C. Cruz, J. González, and D. Pelta. "Optimization in Dynamic Environments: A Survey On Problems, Methods and Measures". In: *Soft Computing* 15 (2011), pp. 1427–1448. DOI: 10.1007/s00500-010-0681-0.

[52] H. Curry. *To H.B. Curry: Essays on Combinatory Logic, Lambda Calculus, and Formalism*. London New York: Academic Press, 1980. ISBN: 978-0-12-349050-6.

[53] L. Dagum and R. Menon. "OpenMP: An Industry Standard API for Shared-Memory Programming". In: *Computational Science & Engineering, IEEE* 5.1 (1998), pp. 46–55.

[54] L. Damas. "Type Assignment in Programming Languages". PhD thesis. University of Edinburgh, 1985.

[55] L. Damas and R. Milner. "Principal Type-Schemes for Functional Programs". In: *Proceedings of the Symposium on Principles of Programming Languages*. 1982, pp. 207–212. DOI: 10.1145/582153.582176.

[56] M. Daneshyari and G. G. Yen. "Dynamic Optimization Using Cultural Based PSO". In: *Proceedings of the IEEE Congress of Evolutionary Computation*. 2011, pp. 509–516. DOI: 10.1109/CEC.2011.5949661.

[57] C. Darwin. *On the Origin of Species by Means of Natural Selection or Preservation of Favoured Races in the Struggle for Life*. 1859.

[58] S. Das and P. N. Suganthan. "Differential Evolution: A Survey of the State-of-the-Art". In: *IEEE Transactions on Evolutionary Computation* 15.1 (2011), pp. 4–31. DOI: 10.1109/TEVC.2010.2059031.

[59] D. Dasgupta. "Advances in Artificial Immune Systems". In: *IEEE Computational Intelligence Magazine* 1.4 (2006), pp. 40–49. DOI: 10.1109/MCI.2006.329705.

[60] D. Dasgupta, ed. *Evolutionary Algorithms in Engineering Applications*. Ed. by Z. Michalewicz. Berlin: Springer-Verlag, 1997.

[61] L. Davis. *Genetic Algorithms and Simulated Annealing*. Research notes in artificial intelligence. Pitman, 1987. ISBN: 9780934613439.

[62] K. De Jong. "Evolving in a Changing World". In: *Proceedings of Foundations of Intelligent Systems*. Ed. by Z. W. Raś and A. Skowron. 1999, pp. 512–519. DOI: 10.1007/BFb0095139.

[63] K. A. De Jong. "An Analysis of the Behavior of a Class of Genetic Adaptive Systems". PhD thesis. USA: University of Michigan, 1975. eprint: https://dl.acm.org/doi/book/10.5555/907087.

[64] K. Deb and H. Beyer. "Self-Adaptive Genetic Algorithms with Simulated Binary Crossover". In: *Evolutionary Computation* 9.2 (2001), pp. 197–221. DOI: 10.1162/106365601750190406.

[65] K. Deb, D. Joshi, and A. Anand. "Real-Coded Evolutionary Algorithms With Parent-Centric Recombination". In: *Proceedings of the IEEE Congress on Evolutionary Computation*. Vol. 1. May 2002, pp. 61–66. DOI: 10.1109/CEC.2002.1006210.

[66] K. Deb. "An Efficient Constraint Handling Method for Genetic Algorithms". In: *Computer Methods in Applied Mechanics and Engineering* 186.2 (2000), pp. 311–338. DOI: 10.1016/S0045-7825(99)00389-8.

[67] K. Deb and D. E. Goldberg. "An Investigation of Niche and Species Formation in Genetic Function Optimization". In: *Proceedings of the International Conference on Genetic Algorithms*. 1989, pp. 42–50.

[68] J. Demšar. "Statistical Comparisons of Classifiers over Multiple Data Sets". In: *Journal of Machine Learning Research* 7 (Dec. 2006), pp. 1–30. DOI: 10.5555/1248547.1248548.

[69] E. Dolstra. "The Purely Functional Software Deployment Model". PhD thesis. Utrecht University, 2006.

[70] C. Dominik. *The Org-Mode 7 Reference Manual: Organize Your Life with GNU Emacs*. with contributions by David O'Toole, Bastien Guerry, Philip Rooke, Dan Davison, Eric Schulte, and Thomas Dye. UK: Network Theory, 2010.

[71] W. Dong *et al.* "Linear Sparse Arrays Designed by Dynamic Constrained Multi-Objective Evolutionary Algorithm". In: *Proceedings of the IEEE Congress on Evolutionary Computation*. 2014, pp. 3067–3072. DOI: 10.1109/CEC.2014.6900448.

[72] M. Dorigo and T. Stützle. *Ant Colony Optimization*. The MIT Press, June 4, 2004. ISBN: 978-0-262-25603-2. DOI: 10.7551/mitpress/1290.001.0001.

[73] M. C. Du Plessis. "Adaptive Multi-Population Differential Evolution for Dynamic Environments". PhD thesis. Pretoria, Gauteng, South Africa: University of Pretoria, 2012.

[74] M. C. du Plessis and A. P. Engelbrecht. "Improved Differential Evolution for Dynamic Optimization Problems". In: *Proceedings of the IEEE Congress on Evolutionary Computation*. 2008, pp. 229–234. DOI: 10.1109/CEC.2008.4630804.

[75] P. Dubois, K. Hinsen, and J. Hugunin. "Numerical Python". In: *Computers in Physics* 10 (May 1996). DOI: 10.1063/1.4822400.

[76] J. G. Duhain. "Particle swarm optimisation in dynamically changing environments-an empirical study". MA thesis. University of Pretoria, 2011.

[77] J. G. Duhain and A. P. Engelbrecht. "Towards a More Complete Classification System For Dynamically Changing Environments". In: *Proceedings of the IEEE Congress on Evolutionary Computation*. IEEE. 2012, pp. 1–8. DOI: 10.1109/CEC.2012.6252881.

[78] J. Durillo, A. Nebro, and E. Alba. "The jMetal Framework for Multi-Objective Optimization: Design and Architecture". In: *Proceedings of the IEEE Congress on Evolutionary Computation*. Vol. 5467. Lecture Notes in Computer Science. July 2010, pp. 4138–4325.

[79] J. J. Durillo and A. J. Nebro. "jMetal: a Java Framework for Multi-Objective Optimization". In: *Advances in Engineering Software* 42 (2011), pp. 760–771. DOI: 10.1016/j.advengsoft.2011.05.014.

[80] J. W. Eaton *et al. GNU Octave Version 5.2.0 Manual: a High-Level Interactive Language for Numerical Computations*. 2020. URL: https://www.gnu.org/software/octave/doc/v5.2.0/.

[81] R. C. Eberhart and Y. Shi. "Tracking and Optimizing Dynamic Systems With Particle Swarms". In: *Proceedings of the IEEE Congress on Evolutionary Computation*. Vol. 1. 2001, pp. 94–100. DOI: 10.1109/CEC.2001.934376.

[82] J. Eggermont, T. Lenaerts, and A. Poyhonen Sannaand Termier. "Raising the Dead: Extending Evolutionary Algorithms with a Case-Based Memory". In: *Proceedings of the European Conference on Genetic Programming*. Ed. by J. Miller *et al.* 2001, pp. 280–290. DOI: 10.1007/3-540-45355-5_22.

[83] M. A. Eita and A. A. Shoukry. "Constrained Dynamic Differential Evolution using a novel hybrid constraint handling technique". In: *Proceedings of the IEEE International Conference on Systems, Man, and Cybernetics*. Oct. 2014, pp. 2421–2426. DOI: 10.1109/SMC.2014.6974289.

[84] A. P. Engelbrecht. *Computational Intelligence: An Introduction*. Second. Chichester, England: Wiley & Sons, Dec. 2007.

[85] A. P. Engelbrecht. *Fundamentals of Computational Swarm Intelligence*. John Wiley & Sons, 2005.

[86] A. P. Engelbrecht. "Roaming Behavior of Unconstrained Particles". In: *Proceedings of the BRICS Countries Congress on Computational Intelligence*. Sept. 2013, pp. 104–111.

[87] D. Espinosa. "Building interpreters by transforming stratified monads". In: *Unpublished manuscript, ftp from altdorf. ai. mit. edu: pub/dae* (1994).

[88] Q. Fan and X. Yan. "Differential Evolution Algorithm with Co-Evolution of Control Parameters and Penalty Factors for Constrained Optimization Problems". In: *Asia-Pacific Journal of Chemical Engineering* 7.2 (2012), pp. 227–235. DOI: 10.1002/apj.524.

[89] Z. Fan *et al.* "A Comparative Study of Constrained Multi-Objective Evolutionary Algorithms on Constrained Multi-Objective Optimization Problems". In: *Proceedings of the IEEE Congress on Evolutionary Computation*. 2017, pp. 209–216. DOI: 10.1109/CEC.2017.7969315.

[90] R. Farmani and J. A. Wright. "Self-Adaptive Fitness Formulation for Constrained Optimization". In: *IEEE Transactions on Evolutionary Computation* 7.5 (2003), pp. 445–455. DOI: 10.1109/TEVC.2003.817236.

[91] W. Feng *et al.* "Benchmarks for Testing Evolutionary Algorithms". In: *Proceedings of the Asia-Pacific Conference on Control and Measurement*. 1998, pp. 134–138. URL: http://eprints.gla.ac.uk/89753/.

[92] C. A. Floudas *et al. Handbook of Test Problems in Local and Global Optimization*. In: *Noncovex Optimization and Its Applications*. Vol. 33. Kluwer Academic Publishers, 1999.

[93] J. N. Foster *et al.* "Combinators for Bi-directional Tree Transformations: A Linguistic Approach to the View Update Problem". In: *ACM SIGPLAN Notices* 40.1 (Jan. 2005), pp. 233–246. DOI: 10.1145/1047659.1040325.

[94] E. Frank, M. A. Hall, and I. H. Witten. *Data Mining: Practical Machine Learning Tools and Techniques*. 4th ed. The WEKA Workbench. Morgan Kaufmann, 2016.

[95] D. Fraser and A. Burnell. *Computer Models in Genetics*. English. 1970. ISBN: 978-0-07-021904-5.

[96] S. García and F. Herrera. "An Extension on "Statistical Comparisons of Classifiers over Multiple Data Sets" for all Pairwise Comparisons". In: *Journal of Machine Learning Research* 9 (Dec. 2008), pp. 2677–2694. URL: https://jmlr.csail.mit.edu/papers/volume9/garcia08a/garcia08a.pdf.

[97] S. García *et al.* "Advanced Nonparametric Tests for Multiple Comparisons in the Design of Experiments In Computational Intelligence and Data Mining: Experimental Analysis of Power". In: *Information Sciences* 180.10 (2010). Special Issue on Intelligent Distributed Information Systems, pp. 2044–2064. DOI: 10.1016/j.ins.2009.12.010.

[98] B. Ghasemishabankareh, X. Li, and M. Ozlen. "Cooperative Coevolutionary Differential Evolution with Improved Augmented Lagrangian to Solve Constrained Optimisation Problems". In: *Information Sciences* 369 (2016), pp. 441–456. DOI: 10.1016/j.ins.2016.06.047.

[99] J.-Y. Girard. "Interprétation fonctionnelle et élimination des coupures de l'arithmétique d'ordre supérieur". PhD thesis. University of Paris, 1972.

[100] F. Glover. "Heuristics for Integer Programming using Surrogate Constraints". In: *Decision Sciences* 8.1 (1977), pp. 156–166. DOI: 10.1111/j.1540-5915.1977.tb01074.x.

[101] C. Goh and K. C. Tan. "A Competitive-Cooperative Coevolutionary Paradigm for Dynamic Multiobjective Optimization". In: *IEEE Transactions on Evolutionary Computation* 13.1 (2009), pp. 103–127. DOI: 10.1109/TEVC.2008.920671.

[102] D. Goldberg and K. Deb. *A Comparative Analysis of Selection Scheme Used In Genetic Algorithms*. Ed. by G. Rawlins. Foundations of Genetic Algorithms. Morgan Kaufmann Publishers, 1991.

[103] W. Gong, Z. Cai, and D. Liang. "Adaptive Ranking Mutation Operator Based Differential Evolution for Constrained Optimization". In: *IEEE Transactions on Cybernetics* 45.4 (2015), pp. 716–727. DOI: 10.1109/TCYB.2014.2334692.

[104] J. J. Grefenstette. "Genetic Algorithms for Changing Environments". In: *Proceedings of the International Conference on Parallel Problem Solving from Nature*. Ed. by R. Maenner and B. Manderick. Vol. 2. 1992, pp. 137–144.

[105] J. J. Grefenstette. *Parallel Adaptive Algorithms for Function Optimization*. Tech. rep. CS-81-19. Vanderbilt University, Computer Science Department, Nashville, 1981.

[106] V. G. Gudise and G. K. Venayagamoorthy. "Comparison of Particle Swarm Optimization and Backpropagation as Training Algorithms for Neural Networks". In: *Proceedings of the IEEE Swarm Intelligence Symposium*. 2003, pp. 110–117. DOI: 10.1109/SIS.2003.1202255.

[107] A. B. Hadj-Alouane and J. C. Bean. "A Genetic Algorithm for the Multiple-Choice Integer Program". In: *Operations Research* 45.1 (1997), pp. 92–101. URL: http://www.jstor.org/stable/171928.

[108] P. Hajela and J. Lee. "Constrained Genetic Search via Schema Adaptation: An Immune Network Solution". In: *Structural optimization* 12 (1 Aug. 1996), pp. 11–15. DOI: 10.1007/BF01270439.

[109] P. Hajela and J. Yoo. "Constraint Handling in Genetic Search Using Expression Strategies". In: *AIAA Journal* 34.11 (1996), pp. 2414–2420. DOI: 10.2514/3.13410.

[110] M. Hall *et al.* "The WEKA Data Mining Software: An Update". In: *Proceedings of the Conference on Knowledge Discovery and Data Mining*. 11 vols. 1. 2009.

[111] R. Hamming. *Error Detecting and Error Codes*. Tech. rep. Vol. XXVI, No. 2. The Bell System Technical Journal, 1950. DOI: 10.1002/j.1538-7305.1950.tb00463.x.

[112] K. R. Harrison, B. M. Ombuki-Berman, and A. P. Engelbrecht. "A Radius-Free Quantum Particle Swarm Optimization Technique for Dynamic Optimization Problems". In: *Proceedings of the IEEE Congress on Evolutionary Computation*. July 2016, pp. 578–585. DOI: 10.1109/CEC.2016.7743845.

[113] K. R. Harrison, B. M. Ombuki-Berman, and A. P. Engelbrecht. "The Effect of Probability Distributions on the Performance of Quantum Particle Swarm Optimization for Solving Dynamic Optimization Problems". In: *Proceedings of the Symposium Series on Computational Intelligence*. Dec. 2015, pp. 242–250. DOI: 10.1109/SSCI.2015.44.

[114] K. Harrison, A. Engelbrecht, and B. Ombuki-Berman. "Self-Adaptive Particle Swarm Optimization: a Review and Analysis of Convergence". In: *Swarm Intellignece* 12 (2018), pp. 187–226. DOI: 10.1007/s11721-017-0150-9.

[115] K. R. Harrison. "An Analysis of Parameter Control Mechanisms for the Particle Swarm Optimization Algorithm". PhD thesis. Pretoria, Gauteng, South Africa: University of Pretoria, 2018.

[116] M. Helbig and A. P. Engelbrecht. "Analysing the Performance of Dynamic Multi-Objective Optimisation Algorithms". In: *Proceedings of the IEEE Congress on Evolutionary Computation*. June 2013, pp. 1531–1539. DOI: 10.1109/CEC.2013.6557744.

[117] M. Helbig and A. P. Engelbrecht. "Issues with Performance Measures for Dynamic Multi-Objective Optimisation". In: *Proceedings of the Symposium on Computational Intelligence in Dynamic and Uncertain Environments*. Apr. 2013, pp. 17–24. DOI: 10.1109/CIDUE.2013.6595767.

[118] R. Hindley. "The Principal Type-Scheme of an Object in Combinatory Logic". In: *Transactions of the American Mathematical Society* 146 (1969), pp. 29–60. URL: http://www.jstor.org/stable/1995158.

[119] F. Hoffmeister and J. Sprave. "Problem-Independent Handling of Constraints by Use of Metric Penalty Functions". In: *Proceedings of the Conference on Evolutionary Programming*. Ed. by L. J. Fogel, P. J. Angeline, and T. Bäck. 1996, pp. 289–294.

[120] J. H. Holland. *Adaption in Natural and Artificial Systems*. University of Michigan Press, 1975.

[121] S. Holm. "A Simple Sequentially Rejective Multiple Test Procedure". In: *Scandinavian Journal of Statistics* 6.2 (1979), pp. 65–70. URL: http://www.jstor.org/stable/4615733.

[122] A. Homaifar, C. X. Qi, and S. H. Lai. "Constrained Optimization Via Genetic Algorithms". In: *SIMULATION* 62.4 (1994), pp. 242–253. DOI: 10.1177/003754979406200405.

[123] X. Hu and R. C. Eberhart. "Tracking Dynamic Systems with PSO: Where's The Cheese". In: *Proceedings of the Workshop on Particle Swarm Optimization*. 2001, pp. 80–83. DOI: 10.1007/s11721-007-0002-0.

[124] D. Huang, R. Gong, and S. Gong. "Constrained Multiobjective Optimization for Microgrid Based on Nondominated Immune Algorithm". In: *IEEE Transactions on Electrical and Electronic Engineering* 10.4 (2015), pp. 376–382. DOI: 10.1002/tee.22096.

[125] F.-z. Huang, L. Wang, and Q. He. "An Effective Co-Evolutionary Differential Evolution for Constrained Optimization". In: *Applied Mathematics and Computation* 186.1 (2007), pp. 340–356. DOI: 10.1016/j.amc.2006.07.105.

[126] J. Hughes. "Software Testing with QuickCheck". In: *Proceedings of the Summer School Conference on Central European Functional Programming*. 2010, pp. 183–223. DOI: 10.5555/1939128.1939134.

[127] S. Janson and M. Middendorf. "A Hierarchical Particle Swarm Optimizer for Noisy and Dynamic Environments". In: *Genetic Programming and Evolvable Machines* 7 (4 2006), pp. 329–354. DOI: 10.1007/s10710-006-9014-6.

[128] D. Jia, S. Qu, and L. Li. "A Multi-swarm Artificial Bee Colony Algorithm for Dynamic Optimization Problems". In: *Proceedings of the International Conference on Information System and Artificial Intelligence*. 2016, pp. 441–445. DOI: 10.1109/ISAI.2016.0100.

[129] Y. Jia *et al.* "Caffe: Convolutional Architecture for Fast Feature Embedding". In: *arXiv preprint arXiv:1408.5093* (2014).

[130] Y. Jin and J. Branke. "Evolutionary Optimization in Uncertain Environments - A Survey". In: *IEEE Transactions on Evolutionary Computation* 9.3 (2005), pp. 303–317. DOI: 10.1109/TEVC.2005.846356.

[131] J. A. Joines and C. R. Houck. "On the Use of Non-Stationary Penalty Functions To Solve Nonlinear Constrained Optimization Problems with GA's". In: *Proceedings of the IEEE Conference on Evolutionary Computation*. Vol. 2. June 1994, pp. 579–584. DOI: 10.1109/ICEC.1994.349995.

[132] M. P. Jones. "Functional Programming with Overloading And Higher-Order Polymorphism". In: *Proceedings of the International School on Advanced Functional Programming*. 1995, pp. 97–136. DOI: 10.1007/3-540-59451-5_4.

[133] A. R. Jordehi. "A Review on Constraint Handling Strategies in Particle Swarm Optimisation". In: *Neural Comput. Appl.* 26.6 (Aug. 2015), pp. 1265–1275. DOI: 10.1007/s00521-014-1808-5.

[134] A. R. Jordehi and J. Jasni. "Parameter Selection in Particle Swarm Optimisation: a Survey". In: *Journal of Experimental & Theoretical Artificial Intelligence* 25.4 (2013), pp. 527–542. DOI: 10.1080/0952813X.2013.782348.

[135] *JSON Schema*. Mar. 2018. URL: https://json-schema.org/ (visited on 03/11/2019).

[136] G. Karafotias, M. Hoogendoorn, and A. E. Eiben. "Parameter Control in Evolutionary Algorithms: Trends and Challenges". In: *IEEE Transactions on Evolutionary Computation* 19.2 (2015), pp. 167–187. DOI: 10.1109/TEVC.2014.2308294.

[137] M. Keijzer *et al.* "Evolving Objects: A General Purpose Evolutionary Computation Library". In: *Artificial Evolution* 2310 (2002), pp. 829–888. URL: http://www.lri.fr/~marc/EO/EO-EA01.ps.gz.

[138] J. Kennedy. "Bare Bones Particle Swarms". In: *Proceedings of the IEEE Swarm Intelligence Symposium*. 2003, pp. 80–87. DOI: 10.1109/SIS.2003.1202251.

[139] J. Kennedy. "Small Worlds and Mega-Minds: Effects of Neighborhood Topology on Particle Swarm Performance". In: *Proceedings of the IEEE Congress on Evolutionary Computation*. Vol. 3. 1999, pp. 1931–1938. DOI: 10.1109/CEC.1999.785509.

[140] J. Kennedy and R. C. Eberhart. "Particle Swarm Optimization". In: *Proceedings of the IEEE International Joint Conference on Neural Networks*. Vol. 4. 1995, pp. 1942–1948. DOI: 10.1109/ICNN.1995.488968.

[141] J. Kennedy and R. Mendes. "Population Structure and Particle Swarm Performance". In: *Proceedings of the IEEE Congress on Evolutionary Computation*. Vol. 2. May 2002, pp. 1671–1676. DOI: 10.1109/CEC.2002.1004493.

[142] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. "Optimization by Simulated Annealing". In: *Science* 220.4598 (1983), pp. 671–680. DOI: 10.1126/science.220.4598.671.

[143] D. E. Knuth. "Literate Programming". In: *The Computer Journal* 27.2 (1984), pp. 97–111. DOI: 10.1093/comjnl/27.2.97.

[144] K. Koppen. "The Curse of Dimensionality". In: *Proceedings of the World Conference on Soft Computing in Industrial Applications*. Sept. 2000.

[145] R. Kowalczyk. "Constraint Consistent Genetic Algorithms". In: *Proceedings of IEEE International Conference on Evolutionary Computation*. 1997, pp. 343–348. DOI: 10.1109/ICEC.1997.592333.

[146] S. Koziel and Z. Michalewicz. "Evolutionary Algorithms, Homomorphous Mappings, and Constrained Parameter Optimization". In: *Evolutionary Computation* 7.1 (1999), pp. 19–44. DOI: 10.1162/evco.1999.7.1.19.

[147] O. Kramer. "A Review of Constraint-Handling Techniques for Evolution Strategies". In: *Applied Computational Intelligence and Soft Computing* 2010 (2010). Ed. by C.-K. Ting, p. 185063. DOI: 10.1155/2010/185063.

[148] E. F. Krause. *Taxicab Geometry: An Adventure in Non-Euclidean Geometry*. Dover Books, Mar. 2019. ISBN: 9780486252025.

[149] R. A. Krohling and L. dos Santos Coelho. "Coevolutionary Particle Swarm Optimization Using Gaussian Distribution for Solving Constrained Optimization Problems". In: *IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics)* 36.6 (2006), pp. 1407–1416. DOI: 10.1109/TSMCB.2006.873185.

[150] W. H. Kruskal and W. A. Wallis. "Use of Ranks in One-Criterion Variance Analysis". In: *Journal of the American Statistical Association* 47.260 (1952), pp. 583–621. URL: http://www.jstor.org/stable/2280779.

[151] V. Kumar. "Algorithms for Constraint Satisfaction Problems: A Survey". In: *A.I. Magazine* 13 (Oct. 1998).

[152] A. Kuri and C. Quezada. "A Universal Eclectic Genetic Algorithm For Constrained Optimization". In: *Proceedings of the European Congress on Intelligent Techniques & Soft Computing*. Jan. 1998.

[153] P. L'Ecuyer and R. Simard. "TestU01: A C Library for Empirical Testing of Random Number Generators". In: *ACM Transactions on Mathematical Software* 33.4 (Aug. 2007), 22:1–22:40. DOI: 10.1145/1268776.1268777.

[154] F. Lardeux and A. Goëffon. "A Dynamic Island-Based Genetic Algorithms Framework". In: *Simulated Evolution and Learning*. Ed. by K. Deb *et al.* Lecture Notes in Computer Science. Berlin, Heidelberg, 2010, pp. 156–165. ISBN: 978-3-642-17298-4. DOI: 10.1007/978-3-642-17298-4_16.

[155] C.-Y. Lee, Z.-J. Lee, and S.-F. Su. "A New Approach for Solving 0/1 Knapsack Problem". In: *Proceedings of the IEEE International Conference on Systems, Man, and Cybernetics*. Vol. 4. 2006, pp. 3138–3143. DOI: 10.1109/ICSMC.2006.384598.

[156] B. J. Leonard and A. P. Engelbrecht. "On the Optimality of Particle Swarm Parameters in Dynamic Environments". In: *Proceedings of the IEEE Congress on Evolutionary Computation*. June 2013, pp. 1564–1569. DOI: 10.1109/CEC.2013.6557748.

[157] X. Leroy *et al. The OCaml system*. Sept. 11, 2019. URL: http://caml.inria.fr/pub/docs/manual-ocaml/.

[158] J. Lewis, E. Hart, and G. Ritchie. "A Comparison of Dominance Mechanisms and Simple Mutation on Non-Stationary Problems". In: *Proceedings of the International Conference on Parallel Problem Solving from Nature*. Ed. by A. E. Eiben *et al.* 1998, pp. 139–148. DOI: 10.1007/BFb0056857.

[159] C. Li. "An Efficient Benchmark Generator for Dynamic Optimization Problems". In: *Proceedings of the International Conference on Bio-Inspired Computing: Theories and Applications*. Ed. by M. Gong *et al.* 2016, pp. 60–72. DOI: 10.1007/978-981-10-3614-9_8.

[160] C. Li and S. Yang. "A Generalized Approach to Construct Benchmark Problems for Dynamic Optimization". In: *Proceedings of the Asia-Pacific Conference on Simulated Evolution and Learning*. Ed. by X. Li *et al.* Nov. 2008, pp. 391–400. DOI: 10.1007/978-3-540-89694-4_40.

[161] C. Li *et al. Benchmark Generator for CEC'2009 Competition on Dynamic Optimization*. Tech. rep. Department of Computer Science, University of Leicester, U.K., Oct. 2008. DOI: 10.13140/RG.2.1.3445.6401.

[162] F. Li and J. Guo. "Topology Optimization of Particle Swarm Optimization". In: *Proceedings of the International Conference on Swarm Intelligence. Advances in Swarm Intelligence*. Ed. by Y. Tan, Y. Shi, and C. Coello. Vol. 8794. Lecture Notes in Computer Science. 2014. DOI: 10.1007/978-3-319-11857-4_16.

[163] W. Li *et al.* "Multipopulation Cooperative Particle Swarm Optimization with a Mixed Mutation Strategy". In: *Information Sciences* 529 (2020), pp. 179–196. DOI: 10.1016/j.ins.2020.02.034.

[164] X. Li *et al.* "Seeking Multiple Solutions: An Updated Survey on Niching Methods and Their Applications". In: *IEEE Transactions on Evolutionary Computation* 21.4 (2017), pp. 518–538. DOI: 10.1109/TEVC.2016.2638437.

[165] J. J. Liang, P. N. Suganthan, and K. Deb. "Novel Composition Test Functions for Numerical Global Optimization". In: *Proceedings of the IEEE Symposium on Swarm Intelligence*. 2005, pp. 68–75. DOI: 10.1109/SIS.2005.1501604.

[166] J. J. Liang, S. Zhigang, and L. Zhihui. "Coevolutionary Comprehensive Learning Particle Swarm Optimizer". In: *Proceedings of the IEEE Congress on Evolutionary Computation*. July 2010, pp. 1–8. DOI: 10.1109/CEC.2010.5585973.

[167] J. Liang *et al. Problem Definitions and Evaluation Criteria for the CEC 2006 Special Session on Constrained Real-Parameter Optimization*. Tech. rep. Nangyang Technological University, Singapore, Jan. 2006.

[168] S. Liang, P. Hudak, and M. Jones. "Monad Transformers and Modular Interpreters". In: *Proceedings of the ACM Symposium on Principles of Programming Languages*. 1995, pp. 333–343. DOI: 10.1145/199448.199528. URL: https://doi.org/10.1145/199448.199528.

[169] T. Lindholm *et al. The Java Language Specification, Java SE 11 Edition*. Sept. 2018. URL: https://docs.oracle.com/javase/specs/jls/se11/jls11.pdf.

[170] B. Liu *et al.* "A Fuzzy Selection Based Constraint Handling Method for Multi-objective Optimization of Analog Cells". In: *Proceedings of the European Conference on Circuit Theory and Design Conference Program*. Sept. 2009, pp. 611–614. DOI: 10.1109/ECCTD.2009.5275048.

[171] C. a. Liu. "New Dynamic Constrained Optimization PSO Algorithm". In: *Proceedings of the International Conference on Natural Computation*. Vol. 7. Oct. 2008, pp. 650–653. DOI: 10.1109/ICNC.2008.742.

[172] M. López-Ibáñez *et al.* "The irace Package: Iterated Racing for Automatic Algorithm Configuration". In: *Operations Research Perspectives* 3 (2016), pp. 43–58. DOI: 10.1016/j.orp.2016.09.002.

[173] S. Luke. *ECJ Evolutionary Computation Library*. Version 27. 1998. URL: http://cs.gmu.edu/~eclab/projects/ecj/.

[174] M. Lunacek and D. Whitley. "The Dispersion Metric and the CMA Evolution Strategy". In: *Proceedings of the Conference on Genetic and Evolutionary Computation*. 2006, pp. 477–484. DOI: 10.1145/1143997.1144085.

[175] X. Ma *et al.* "A Survey on Cooperative Co-Evolutionary Algorithms". In: *IEEE Transactions on Evolutionary Computation* 23.3 (2019), pp. 421–441. DOI: 10.1109/TEVC.2018.2868770.

[176] G. Madey *et al.* "Agent-Based Scientific Simulation". In: *Computing in Science & Engineering* 2.01 (Jan. 2005), pp. 22–29. DOI: 10.1109/MCSE.2005.7.

[177] K. M. Malan. "Characterising Continuous Optimisation Problems for Particle Swarm Optimisation Performance Prediction". PhD thesis. University of Pretoria, 2014.

[178] K. M. Malan and A. P. Engelbrecht. "Characterising the Searchability of Continuous Optimisation Problems for PSO". In: *Swarm Intelligence* 8.4 (Dec. 2014), pp. 275–302. DOI: 10.1007/s11721-014-0099-x.

[179] K. M. Malan and I. Moser. "Constraint Handling Guided by Landscape Analysis in Combinatorial and Continuous Search Spaces". In: *Evolutionary Computation* (Mar. 2018), pp. 1–23. DOI: 10.1162/evco_a_00222.

[180] K. M. Malan, J. F. Oberholzer, and A. P. Engelbrecht. "Characterising Constrained Continuous Optimisation Problems". In: *Proceedings of the IEEE Congress on Evolutionary Computation*. May 2015, pp. 1351–1358. DOI: 10.1109/CEC.2015.7257045.

[181] K. M. Malan and A. P. Engelbrecht. "A Survey of Techniques for Characterising Fitness Landscapes and Some Possible Ways Forward". In: *Information Sciences* 241 (2013), pp. 148–163. DOI: 10.1016/j.ins.2013.04.015.

[182] R. Mallipeddi and P. Suganthan. *Problem Definitions and Evaluation Criteria for The CEC 2010 Competition on Constrained Real-Parameter Optimization*. Tech. rep. Nanyang Technological University, Singapore, May 2010.

[183] S. Marlow *et al. Haskell 2010 Language Report*. July 2010. URL: https://www.haskell.org/definition/haskell2010.pdf.

[184] G. Marsaglia. *The Marsaglia Random Number CDROM, with The Diehard Battery of Tests of Randomness*. CD-ROM. Produced at Florida State University under a grant from The National Science Foundation, 1985. Access available at http://www.stat.fsu.edu/pub/diehard. Florida State University, 1985.

[185] Martín Abadi *et al.* *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*. White paper. https://www.tensorflow.org/, 2015.

[186] M. Mavrovouniotis, C. Li, and S. Yang. "A Survey of Swarm Intelligence for Dynamic Optimization: Algorithms and Applications". In: *Swarm and Evolutionary Computation* 33 (2017), pp. 1–17. DOI: 10.1016/j.swevo.2016.12.005.

[187] C. McBride and R. Paterson. "Applicative Programming with Effects". In: *Journal of Functional Programming* 18.1 (2008), pp. 1–13. DOI: 10.1017/S0956796807006326.

[188] S. Melnik *et al.* "Dremel: Interactive Analysis of Web-Scale Datasets". In: *Proceedings of the International Conference on Very Large Data Bases*. 2010, pp. 330–339. URL: http://www.vldb2010.org/accept.htm.

[189] R. Mendes and A. S. Mohais. "DynDE: a Differential Evolution for Dynamic Optimization Problems". In: *Proceedings of the IEEE Congress on Evolutionary Computation*. Vol. 3. 2005, pp. 2808–2815. DOI: 10.1109/CEC.2005.1555047.

[190] D. Merkel. "Docker: Lightweight Linux Containers for Consistent Development and Deployment". In: *Linux Journal* 2014.239 (Mar. 2014).

[191] Z. Michalewicz and G. Nazhiyath. "Genocop III: a Co-Evolutionary Algorithm For Numerical Optimization Problems with Nonlinear Constraints". In: *Proceedings of IEEE International Conference on Evolutionary Computation*. Vol. 2. 1995, pp. 647–651.

[192] Z. Michalewicz and N. F. Attia. "Evolutionary Optimization of Constrained Problems". In: *Proceedings of the Conference on Evolutionary Programming*. 1994, pp. 98–108.

[193] D. MICHIE. ""Memo" Functions and Machine Learning". In: *Nature* 218.5136 (1968), pp. 19–22. DOI: 10.1038/218019a0.

[194] J. H. Miller. "The Coevolution of Automata in the Repeated Prisoner's Dilemma". In: *Journal of Economic Behavior & Organization* 29.1 (1996), pp. 87–112. DOI: 10.1016/0167-2681(95)00052-6.

[195] K. Millman and M. Aivazis. "Python for Scientists and Engineers". In: *Computing in Science & Engineering* 13.02 (Mar. 2011), pp. 9–12. DOI: 10.1109/MCSE.2011.36.

[196] R. Milner. "A Theory of Type Polymorphism in Programming". In: *Journal of Computer and System Sciences* 17.3 (1978), pp. 348–375. DOI: 10.1016/0022-0000(78)90014-4.

[197] E. Moggi. "Notions of Computation and Monads". In: *Information and Computation* 93.1 (1991). Selections from 1989 IEEE Symposium on Logic in Computer Science, pp. 55–92. DOI: 10.1016/0890-5401(91)90052-4.

[198] C. K. Monson and K. D. Seppi. "Linear Equality Constraints and Homomorphous Mappings in PSO". In: *Proceedings of the IEEE Congress on Evolutionary Computation*. Vol. 1. 2005, pp. 73–80. DOI: 10.1109/CEC.2005.1554669.

[199] R. W. Morrison and K. A. De Jong. "A Test Problem Generator for Non-Stationary Environments". In: *Proceedings of the IEEE Congress on Evolutionary Computation*. Vol. 3. 1999, pp. 2047–2053.

[200] R. W. Morrison. "Performance Measurement in Dynamic Environments". In: *Proceedings of the Bird of a Feather Workshops, Genetic and Evolutionary Computation Conference*. Ed. by A. M. Barry. Aug. 2003, pp. 99–102.

[201] I. Moser and R. Chiong. "Dynamic Function Optimization: The Moving Peaks Benchmark". In: *Metaheuristics for Dynamic Optimization*. Ed. by E. Alba, A. Nakib, and P. Siarry. Springer Berlin Heidelberg, 2013, pp. 35–59. ISBN: 978-3-642-30665-5. DOI: 10.1007/978-3-642-30665-5_3.

[202] H. Mühlenbein and D. Schlierkamp-Voosen. "Predictive Models for the Breeder Genetic Algorithm". In: *Evolutionary Computation*. Continuous Parameter Optimization 1 (1993), pp. 25–49.

[203] *mypy: Optional static type checking for Python*. Dec. 1, 2018. URL: http://www.mypy-lang.org/.

[204] M. Neethling and A. P. Engelbrecht. "Determining RNA Secondary Structure using Set-based Particle Swarm Optimization". In: *Proceedings of the IEEE International Conference on Evolutionary Computation*. 2006, pp. 1670–1677. DOI: 10.1109/CEC.2006.1688509.

[205] C. Negus. *Docker Containers*. 2nd. Addison-Wesley Professional, 2015. ISBN: 9780134397511.

[206] T. T. Nguyen. "Continuous Dynamic Optimisation Using Evolutionary Algorithms". PhD thesis. Department of Computer Science, University of Birmingham, 2011.

[207] T. T. Nguyen and X. Yao. "Benchmarking and Solving Dynamic Constrained Problems". In: *Proceedings of the IEEE Congress on Evolutionary Computation*. May 2009, pp. 690–697. DOI: 10.1109/CEC.2009.4983012.

[208] T. T. Nguyen and X. Yao. "Continuous Dynamic Constrained Optimization - The Challenges". In: *IEEE Transactions on Evolutionary Computation* 16.6 (Dec. 2012), pp. 769–786. DOI: 10.1109/TEVC.2011.2180533.

[209] T. T. Nguyen, S. Yang, and J. Branke. "Evolutionary Dynamic Optimization: a Survey of The State of the Art". In: *Swarm and Evolutionary Computation* 6 (2012), pp. 1–24. DOI: 10.1016/j.swevo.2012.05.001.

[210] T. T. Nguyen and X. Yao. "Dynamic Time-Linkage Problems Revisited". In: *Proceedings of the Workshops on Applications of Evolutionary Computation*. Ed. by M. Giacobini *et al.* 2009, pp. 735–744. DOI: 10.1007/978-3-642-01129-0_83.

[211] T. T. Nguyen and X. Yao. *Solving Dynamic Constrained Optimisation Problems Using Repair Methods*. 2010.

[212] P. Novoa-Hernández, C. Carlos Alberto, and D. Pelta. "Self-Adaptation in Dynamic Environments - a Survey and Open Issues". In: *International Journal of Bio-Inspired Computation* 8 (Jan. 2016), pp. 1–13. DOI: 10.1504/IJBIC.2016.074635.

[213] B. O'Sullivan, D. Stewart, and J. Goerzen. *Real World Haskell*. O'Reilly, 2008. 700 pp. ISBN: 978-0596514983.

[214] M. Odersky and al. *An Overview of the Scala Programming Language*. Tech. rep. IC/2004/64. Lausanne, Switzerland: EPFL, 2004.

[215] C. Okasaki. "Purely Functional Data Structures". PhD thesis. Carnegie Mellon University, 1996.

[216] C. Okasaki. *Purely Functional Data Structures*. Cambridge: Cambridge University Press, 1998. DOI: 10.1017/CBO9780511530104.

[217] T. Oliphant. "Python for Scientific Computing". In: *Computing in Science & Engineering* 9 (June 2007), pp. 10–20. DOI: 10.1109/MCSE.2007.58.

[218] O. Olorunda and A. P. Engelbrecht. "Measuring Exploration/Exploitation in Particle Swarms Using Swarm Diversity". In: *Proceedings of the IEEE Congress on Evolutionary Computation*. 2008, pp. 1128–1134. DOI: 10.1109/CEC.2008.4630938.

[219] A. L. Olsen. "Penalty Functions for the Knapsack Problem". In: *Proceedings of the IEEE Conference on Evolutionary Computation*. Vol. 2. 1994, pp. 554–558.

[220] Oracle. *Javadoc Tool*. Version 5.0. 2019. URL: https://docs.oracle.com/javase/1.5.0/docs/guide/javadoc/index.html.

[221] G. Pamparà and A. P. Engelbrecht. "Self-adaptive Quantum Particle Swarm Optimization for Dynamic Environments". In: *Proceedings of the International Conference on Swarm Intelligence*. Ed. by M. Dorigo *et al.* 2018, pp. 163–175. DOI: 10.1007/978-3-030-00533-7_13.

[222] G. Pamparà and A. P. Engelbrecht. "Towards A Generic Computational Intelligence Library: Preventing Insanity". In: *Proceedings of the IEEE Symposium Series on Computational Intelligence*. 2015 IEEE Symposium Series on Computational Intelligence. Dec. 2015, pp. 1460–1467. DOI: 10.1109/SSCI.2015.207.

[223] G. Pamparà, A. Engelbrecht, and T. Cloete. "CIlib: a Collaborative Framework for Computational Intelligence Algorithms - Part I". In: *Proceedings of the IEEE International Joint Conference on Neural Networks*. June 2008, pp. 1750–1757. DOI: 10.1109/IJCNN.2008.4634035.

[224] G. Pamparà, F. Nepomuceno, and B. Leonard. *CIlib v2.0.1*. Oct. 2014. DOI: 10.5281/zenodo.12371.

[225] G. Pamparà and A. P. Engelbrecht. "A Generator for Dynamically Constrained Optimization Problems". In: *Proceedings of the Genetic and Evolutionary Computation Conference Companion*. GECCO '19. July 13, 2019, pp. 1441–1448. DOI: 10.1145/3319619.3326798.

[226] G. Pamparà and A. P. Engelbrecht. "Evolutionary and Swarm Intelligence Algorithms Through Monadic Composition". In: *Proceedings of the Genetic and Evolutionary Computation Conference Companion*. GECCO '19. July 13, 2019, pp. 1382–1390. DOI: 10.1145/3319619.3326845.

[227] J. Paredis. "Co-Evolutionary Constraint Satisfaction". In: *Proceedings of the International Conference of Parallel Problem Solving from Nature*. Ed. by Y. Davidor, H.-P. Schwefel, and R. Männer. 1994, pp. 46–55.

[228] A. Paszke *et al.* "PyTorch: An Imperative Style, High-Performance Deep Learning Library". In: *Advances in Neural Information Processing Systems*. Ed. by H. Wallach *et al.* Curran Associates, Inc., 2019, pp. 8024–8035. URL: http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf.

[229] E. Peer. "A Serendipitous Software Framework for Facilitating Collaboration in Computational Intelligence". MA thesis. University of Pretoria, 2005.

[230] E. Peer *et al.* "CiClops: Computational Intelligence Collaborative Laboratory of Pantological Software". In: *Proceedings of the IEEE Swarm Intelligence Symposium*. June 2005, pp. 130–137. DOI: 10.1109/SIS.2005.1501612.

[231] E. Peer *et al.* "CiClops: Computational Intelligence Collaborative Laboratory of Pantological Software". In: *Proceedings of the IEEE Swarm Intelligence Symposium*. 2005.

[232] B. Pierce *et al. Basic Category Theory for Computer Scientists*. Foundations of computing series: research reports and notes. MIT Press, 1991. ISBN: 9780262660716. URL: https://books.google.co.za/books?id=ezdeaHfpYPwC.

[233] E. Pitzer and M. Affenzeller. "A Comprehensive Survey on Fitness Landscape Analysis". In: *Recent Advances in Intelligent Engineering Systems*. Vol. 378. Berlin, Germany, Oct. 2012, pp. 161–191. DOI: 10.1007/978-3-642-23229-9_8.

[234] R. Poli. "Mean and Variance of the Sampling Distribution of Particle Swarm Optimizers During Stagnation". In: *IEEE Transactions on Evolutionary Computation* 13.4 (2009), pp. 712–721. DOI: 10.1109/TEVC.2008.2011744.

[235] M. A. Potter and K. A. De Jong. "A Cooperative Coevolutionary Approach to Function Optimization". In: *Proceedings of the International Conference on Parallel Problem Solving from Nature*. Ed. by Y. Davidor, H.-P. Schwefel, and R. Männer. 1994, pp. 249–257. DOI: 10.1007/3-540-58484-6_269.

[236] T. Preston-Werner. *Semantic Versioning 2.0.0*. 2013. URL: http://semver.org.

[237] A. K. Qin, V. L. Huang, and P. N. Suganthan. "Differential Evolution Algorithm With Strategy Adaptation for Global Numerical Optimization". In: *IEEE Transactions on Evolutionary Computation* 13.2 (2009), pp. 398–417. DOI: 10.1109/TEVC.2008.927706.

[238] A. K. Qin and P. N. Suganthan. "Self-Adaptive Differential Evolution Algorithm For Numerical Optimization". In: *Proceedings of the IEEE Congress on Evolutionary Computation*. Vol. 2. 2005, pp. 1785–1791. DOI: 10.1109/CEC.2005.1554904.

[239] S. Rajasekaran and S. Lavanya. "Hybridization of Genetic Algorithms with Immune System for Optimization Problems in Structural Engineering". In: *Structural and Multidisciplinary Optimization* 34.5 (2007), pp. 415–429.

[240] J. Rapin and O. Teytaud. *Nevergrad - A Gradient-Free Optimization Platform*. Version 0.4.2. 2018. URL: https://github.com/FacebookResearch/Nevergrad.

[241] J. C. Reynolds. "Towards a Theory of Type Structure". In: *Proceedings of the Symposium on Programming*. Ed. by B. Robinet. 1974, pp. 408–425. DOI: 10.1007/3-540-06859-7_148.

[242] R. G. Reynolds. "An Introduction to Cultural Algorithms". In: *Proceedings of the Conference on Evolutionary Programming*. 1994, pp. 131–139.

[243] J. T. Richardson *et al.* "Some Guidelines for Genetic Algorithms with Penalty Functions". In: *Proceedings of the International Conference on Genetic Algorithms*. 1989, pp. 191–197.

[244] H. Richter. "Detecting Change in Dynamic Fitness Landscapes". In: *Proceedings of the IEEE Congress on Evolutionary Computation*. IEEE. 2009, pp. 1613–1620. DOI: 10.1109/CEC.2009.4983135.

[245] H. Richter. "Memory Design for Constrained Dynamic Optimization Problems". In: *Proceedings of the Applications of Evolutionary Computation*. Ed. by C. Di Chio *et al.* 2010, pp. 552–561. DOI: 10.1007/978-3-642-12239-2_57.

[246] M. Riekert, K. M. Malan, and A. P. Engelbrecht. "Adaptive Genetic Programming for Dynamic Classification Problems". In: *Proceedings of the IEEE Congress on Evolutionary Computation*. 2009, pp. 674–681. DOI: 10.1109/CEC.2009.4983010.

[247] S. Roman. *Coding and Information Theory*. 1st ed. Vol. 134. Graduate Texts in Mathematics, Number Theory. New York: Springer-Verlag, 1992. 488 pp. ISBN: 978-0-387-97812-3.

[248] C. Rossi, M. Abderrahim, and J. C. Díaz. "Tracking Moving Optima Using Kalman-Based Predictions". In: *Evolutionary Computation* 16.1 (2008), pp. 1–30. DOI: 10.1162/evco.2008.16.1.1.

[249] T. P. Runarsson and X. Yao. "Stochastic Ranking for Constrained Evolutionary Optimization". In: *IEEE Transactions on Evolutionary Computation* 4.3 (2000), pp. 284–294. DOI: 10.1109/4235.873238.

[250] C. Saha *et al.* "A Fuzzy Rule-Based Penalty Function Approach for Constrained Evolutionary Optimization". In: *IEEE Transactions on Cybernetics* 46.12 (2016), pp. 2953–2965. DOI: 10.1109/TCYB.2014.2359985.

[251] S. Salcedo-Sanz. "A Survey of Repair Methods Used as Constraint Handling Techniques in Evolutionary Algorithms". In: *Computer Science Review* 3.3 (2009), pp. 175–192. DOI: 10.1016/j.cosrev.2009.07.001.

[252] S. Saleem and R. Reynolds. "Cultural Algorithms in Dynamic Environments". In: *Proceedings of the IEEE Congress on Evolutionary Computation*. Vol. 2. 2000, pp. 1513–1520. DOI: 10.1109/CEC.2000.870833.

[253] E. Schulte and D. Davison. "Active Documents with Org-Mode". In: *Computing in Science Engineering* 13.3 (May 2011), pp. 66–73. DOI: 10.1109/MCSE.2011.41.

[254] E. Schulte *et al.* "A Multi-Language Computing Environment for Literate Programming and Reproducible Research". In: *Journal of Statistical Software* 46.3 (Jan. 2012), pp. 1–24. URL: http://www.jstatsoft.org/v46/i03.

[255] F. Seide and A. Agarwal. "CNTK: Microsoft's Open-Source Deep-Learning Toolkit". In: *Proceedings of the ACM International Conference on Knowledge Discovery and Data Mining*. 2016, p. 2135. DOI: 10.1145/2939672.2945397.

[256] Y. Shi and R. A. Krohling. "Co-Evolutionary Particle Swarm Optimization To Solve Min-Max Problems". In: *Proceedings of the IEEE Congress on Evolutionary Computation*. Vol. 2. 2002, pp. 1682–1687. DOI: 10.1109/CEC.2002.1004495.

[257] W. Siedlecki and J. Sklansky. "Constrained Genetic Optimization via Dynamic Reward-Penalty Balancing and its use in Pattern Recognition". In: *Handbook of Pattern Recognition and Computer Vision*. 1993, pp. 108–123. DOI: 10.1142/9789814343138_0006.

[258] A. Simões and E. Costa. "Evolutionary Algorithms for Dynamic Environments: Prediction Using Linear Regression and Markov Chains". In: *Proceedings of the International Conference on Parallel Problem Solving from Nature*. Ed. by G. Rudolph *et al.* 2008, pp. 306–315. DOI: 10.1007/978-3-540-87700-4_31.

[259] A. Simões and E. Costa. "Improving Prediction in Evolutionary Algorithms For Dynamic Environments". In: *Proceedings of the Genetic and Evolutionary Computation Conference*. Ed. by F. Rothlauf. 2009, pp. 875–882. DOI: 10.1145/1569901.1570021.

[260] A. E. Smith and D. M. Tate. "Genetic Optimization Using A Penalty Function". In: *Proceedings of the International Conference on Genetic Algorithms*. 1993, pp. 499–505. DOI: 10.5555/645513.657589.

[261] A. E. Smith and D. M. Coit. "Constraint Handling Techniques – Penalty Functions". In: *Handbook of Evolutionary Computation*. Ed. by T. Bäck, D. B. Fogel, and Z. Michalewicz. Oxford University Press and Institute of Physics Publishing, 1997. Chap. C 5.2.

[262] A. E. Smith and D. M. Tate. "Genetic Optimization Using A Penalty Function". In: *Proceedings of the International Conference on Genetic Algorithms*. 1993, pp. 499–505. DOI: 10.5555/645513.657589.

[263] M. Spiegel, J. Schiller, and R. Srinivasan. *Probability and Statistics*. 4th ed. Schaum's Outline. McGraw-Hill Education, 2012. ISBN: 978-0071795579.

[264] G. L. Steele. "Building interpreters by composing monads". In: *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '94. New York, NY, USA: Association for Computing Machinery, Feb. 1, 1994, pp. 472–492. ISBN: 978-0-89791-636-3. DOI: 10.1145/174675.178068.

[265] S. van der Stockt and A. P. Engelbrecht. "Analysis of Hyper-Heuristic Performance in Different Dynamic Environments". In: *Proceedings of the Symposium on Computational Intelligence in Dynamic and Uncertain Environments*. IEEE. 2014, pp. 1–8. DOI: 10.1109/CIDUE.2014.7007860.

[266] R. Storn. "On the Usage of Differential Evolution for Function Optimization". In: *Proceedings of North American Fuzzy Information Processing*. 1996, pp. 519–523. DOI: 10.1109/NAFIPS.1996.534789.

[267] R. Storn and K. Price. "Differential Evolution – A Simple and Efficient Heuristic for Global Optimization over Continuous Spaces". In: *Journal of Global Optimization* 11.4 (Dec. 1997), pp. 341–359. DOI: 10.1023/A:1008202821328.

[268] R. Sturm and T. Frudakis. "Eye Colour: Portals into Pigmentation Genes And Ancestry". In: *Trends in Genetics* 20.8 (2004), pp. 327–332. DOI: 10.1016/j.tig.2004.06.010.

[269] P. Suganthan *et al.* "Problem Definitions and Evaluation Criteria for the CEC 2005 Special Session on Real-Parameter Optimization". In: *Natural Computing* (Jan. 2005), pp. 341–357.

[270] M.-J. Tahk and B.-C. Sun. "Coevolutionary Augmented Lagrangian Methods For Constrained Optimization". In: *IEEE Transactions on Evolutionary Computation* 4.2 (2000), pp. 114–124. DOI: 10.1109/4235.850652.

[271] T. Takahama and S. Sakai. "Constrained Optimization by Applying the α Constrained Method to the Nonlinear Simplex Method with Mutations". In: *IEEE Transactions on Evolutionary Computation* 9.5 (2005), pp. 437–451. DOI: 10.1109/TEVC.2005.850256.

[272] T. Takahama and S. Sakai. "Solving Constrained Optimization Problems by the ϵ-Constrained Particle Swarm Optimizer with Adaptive Velocity Limit Control". In: *Proceedings of the IEEE Conference on Cybernetics and Intelligent Systems*. 2006, pp. 1–7. DOI: 10.1109/ICCIS.2006.252248.

[273] T. Takahama and S. Sakai. "Constrained Optimization by Combining the α Constrained Method with Particle Swarm Optimization". In: *Proceedings of Joint International Conference on Soft Computing and Intelligent Systems and the International Symposium on Advanced Intelligent Systems*. 2004.

[274] R. C. Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing. Vienna, Austria, 2013. URL: http://www.R-project.org/.

[275] T. C. D. Team. *The Coq Proof Assistant, version 8.10.0*. Version 8.10.0. Oct. 2019. DOI: 10.5281/zenodo.3476303.

[276] T. D. Team. "Theano: A Python framework for fast computation of mathematical expressions". In: *arXiv e-prints* abs/1605.02688 (May 2016). URL: http://arxiv.org/abs/1605.02688.

[277] A. Toffolo and E. Benini. "Genetic Diversity as an Objective in Multi-Objective Evolutionary Algorithms". In: *Evolutionary Computation* 11.2 (2003), pp. 151–167. DOI: 10.1162/106365603766646816.

[278] I. C. Trelea. "The particle swarm optimization algorithm: convergence analysis and parameter selection". In: *Information Processing Letters* 85.6 (2003), pp. 317–325. DOI: 10.1016/S0020-0190(02)00447-7.

[279] K. Trojanowski and Z. Michalewicz. "Searching for Optima in Non-Stationary Environments". In: *Proceedings of the IEEE Congress on Evolutionary Computation*. Vol. 3. 1999, pp. 1843–1850. DOI: 10.1109/CEC.1999.785498.

[280] D. A. Umbarkar and P. Sheth. "Crossover Operators in Genetic Algorithms: A Review". In: *ICTACT Journal on Soft Computing* 6 (Oct. 2015). DOI: 10.21917/ijsc.2015.0150.

[281] T. Univalent Foundations Program. *Homotopy Type Theory: Univalent Foundations of Mathematics*. Institute for Advanced Study, 2013. URL: https://homotopytypetheory.org/book.

[282] R. K. Ursem *et al.* "Analysis and Modeling of Control Tasks in Dynamic Systems". In: *IEEE Transactions on Evolutionary Computation* 6.4 (Aug. 2002), pp. 378–389. DOI: 10.1109/TEVC.2002.802871.

[283] R. Ursem. "Multinational GA optimization techniques in dynamic environments". In: *Proceedings of the Genetic and Evolutionary Computation Conference*. Ed. by D. Whitley *et al.* 2000, pp. 19–26.

[284] F. van den Bergh and A. P. Engelbrecht. "A Cooperative Approach to Particle Swarm Optimization". In: *Transactions on Evolutionary Computation* 8.3 (2004), pp. 225–239. DOI: 10.1109/TEVC.2004.826069.

[285] F. Van Den Bergh. "An Analysis of Particle Swarm Optimizers". PhD thesis. Pretoria, South Africa, South Africa, 2002.

[286] T. Van Le. "A Fuzzy Evolutionary Approach to Constrained Optimisation Problems". In: *Proceedings of IEEE International Conference on Evolutionary Computation*. 1996, pp. 274–278. DOI: 10.1109/ICEC.1996.542374.

[287] S. van Rijn *et al.* "Optimizing Highly Constrained Truck Loadings Using A Self-Adaptive Genetic Algorithm". In: *Proceedings of the IEEE Congress on Evolutionary Computation*. 2015, pp. 227–234. DOI: 10.1109/CEC.2015.7256896.

[288] V. K. Vassilev. "Fitness Landscapes and Search in the Evolutionary Design of Digital Circuits". PhD thesis. Napier University, 2000.

[289] V. K. Vassilev, T. C. Fogarty, and J. F. Miller. "Information Characteristics and the Structure of Landscapes". In: *Evolutionary Computation* 8.1 (Mar. 2000), pp. 31–60. DOI: 10.1162/106365600568095.

[290] V. K. Vassilev, T. C. Fogarty, and J. F. Miller. "Smoothness, Ruggedness and Neutrality of Fitness Landscapes: from Theory to Application". In: *Advances in Evolutionary Computing: Theory and Applications*. Ed. by A. Ghosh and S. Tsutsui. Springer Berlin Heidelberg, 2003, pp. 3–44. ISBN: 978-3-642-18965-4. DOI: 10.1007/978-3-642-18965-4_1.

[291] S. Vickers. *Topology via Logic (Cambridge Tracts in Theoretical Computer Science)*. Cambridge University Press, 1996. ISBN: 9780521576512.

[292] H. Voigt and H. Muhlenbein. "Gene Pool Recombination and Utilization Of Covariances for the Breeder Genetic Algorithm". In: *Proceedings of IEEE International Conference on Evolutionary Computation*. Vol. 1. 1995, pp. 172–. DOI: 10.1109/ICEC.1995.489139.

[293] P. Wadler. "Comprehending Monads". In: *Proceedings of the ACM Conference on LISP and Functional Programming*. 1990, pp. 61–78. DOI: 10.1145/91556.91592. URL: http://doi.acm.org/10.1145/91556.91592.

[294] P. Wadler. "Monads for Functional Programming". In: *Proceedings of the Advanced Functional Programming, First International Spring School on Advanced Functional Programming Techniques-Tutorial Text*. 1995, pp. 24–52. URL: http://dl.acm.org/citation.cfm?id=647698.734146.

[295] S. van der Walt, S. C. Colbert, and G. Varoquaux. "The NumPy Array: A Structure for Efficient Numerical Computation". In: *Computing in Science and Engineering* 13.2 (Mar. 2011). Provided by the SAO/NASA Astrophysics Data System, pp. 22–30. DOI: 10.1109/MCSE.2011.37.

[296] Y. Wang and M. Wineberg. "Estimation of Evolvability Genetic Algorithm And Dynamic Environments". In: *Genetic Programming and Evolvable Machines volume* 7 (4 2006), pp. 355–382. DOI: 10.1007/s10710-006-9015-5.

[297] K. Weicker. "An Analysis of Dynamic Severity and Population Size". In: *Proceedings of the International Conference on Parallel Problem Solving from Nature*. Ed. by M. Schoenauer *et al.* 2000, pp. 159–168. DOI: 10.1007/3-540-45356-3_16.

[298] K. Weicker. "Evolutionary Algorithms and Dynamic Optimization Problem". PhD thesis. Der Andere Verlag, 2003.

[299] K. Weicker. "Performance Measures for Dynamic Environments". In: *Proceedings of the Conference on Parallel Problem Solving from Nature*. Ed. by J. J. M. Guervós *et al.* 2002, pp. 64–73. DOI: 10.1007/3-540-45712-7_7.

[300] K. Weicker and N. Weicker. "Dynamic Rotation and Partial Visibility". In: *Proceedings of the IEEE Congress on Evolutionary Computation*. Vol. 2. 2000, pp. 1125–1131. DOI: 10.1109/CEC.2000.870774.

[301] J. Wells. "Typability and Type Checking in System F Are Equivalent and Undecidable". In: *Annals of Pure and Applied Logic* 98.1 (1999), pp. 111–156. DOI: 10.1016/S0168-0072(98)00047-5.

[302] D. Whitley, V. Gordon, and K. Mathias. "Lamarckian Evolution, the Baldwin Effect And Function Optimization". In: *Parallel Problem Solving from Nature*. Vol. 866. Springer, Berlin, Heidelberg, 1994, pp. 5–15. ISBN: 978-3-540-58484-1. DOI: 10.1007/3-540-58484-6_245.

[303] D. Whitley, S. Rana, and R. B. Heckendorn. "The Island Model Genetic Algorithm: On Separability, Population Size and Convergence". In: *Journal of computing and information technology* 7.1 (1999), pp. 33–47.

[304] D. H. Wolpert and W. G. Macready. "No Free Lunch Theorems for Optimization". In: *IEEE Transactions on Evolutionary Computation* 1.1 (1997), pp. 67–82. DOI: 10.1109/4235.585893.

[305] B. Wu, X. Yu, and L. Liu. "Fuzzy Penalty Function Approach for Constrained Function Optimization with Evolutionary Algorithms". In: *Proceedings of the International Conference on Neural Information Processing*. Mar. 2003.

[306] G. Wu, R. Mallipeddi, and P. Suganthan. *Problem Definitions and Evaluation Criteria for the CEC 2017 Competition and Special Session on Constrained Single Objective Real-Parameter Optimization*. Tech. rep. Nanyang Technological University, Singapore, Oct. 2016.

[307] Y. Xie. *Dynamic Documents with R and knitr*. 2nd. Boca Raton, Florida: Chapman and Hall/CRC, 2015. ISBN: 978-1-498-71696-3. URL: https://yihui.name/knitr/.

[308] Y. Xie. "knitr: A Comprehensive Tool for Reproducible Research in R". In: *Implementing Reproducible Computational Research*. Ed. by V. Stodden, F. Leisch, and R. D. Peng. Chapman and Hall/CRC, 2014. ISBN: 978-1-466-56159-5. URL: http://www.crcpress.com/product/isbn/9781466561595.

[309] Y. Xie. *knitr: A General-Purpose Package for Dynamic Report Generation in R*. R package version 1.20. 2018. URL: https://yihui.name/knitr/.

[310] S. Yang and X. Yao. "Experimental Study on Population-Based Incremental Learning Algorithms for Dynamic Optimization Problems". In: *Soft Computing* 9 (11 2005), pp. 815–834. DOI: 10.1007/s00500-004-0422-3.

[311] Y. Yin and L. Sun. "Generalized Dynamic Constraint Satisfaction Based on Extension Particle Swarm Optimization Algorithm for Collaborative Simulation". In: *Proceedings of the IEEE International Conference on Computer-Aided Design and Computer Graphics*. Oct. 2007, pp. 541–544. DOI: 10.1109/CADCG.2007.4407950.

[312] B. Yorgey. "The Typeclassopedia". In: *The Monad Reader* (13 2009), p. 17.

[313] T. Yu and C. Clack. "PolyGP: A Polymorphic Genetic Programming System In Haskell". In: *Proceedings of the Genetic Programming Conference*. Jan. 1998, pp. 416–421. eprint: https://discovery.ucl.ac.uk/id/eprint/10087128.

[314] M. Zaharia *et al.* "Spark: Cluster Computing with Working Sets". In: *Proceedings of the USENIX Conference on Hot Topics in Cloud Computing*. 2010. eprint: https://dl.acm.org/doi/10.5555/1863103.1863113.

[315] S. Zeng *et al.* "A General Framework of Dynamic Constrained Multiobjective Evolutionary Algorithms for Constrained Optimization". In: *IEEE Transactions on Cybernetics* 47.9 (Sept. 2017), pp. 2678–2688. DOI: 10.1109/TCYB.2017.2647742.

Appendix A

Algorithm Rankings Across Problem Benchmark Instances

Table A.1: Algorithm wins and losses for each CMPB benchmark problem instance

	Objective Space Behaviour	Constraint Space Behaviour	RIGA	HyperM	DDECV	CCSaQPSO	SaQPSO	CCSaDE	SaDE	CCRIGA	CCHyperM	CCGVPSO	GVPSO	CCSaQGVPSO	Problem Changes
1	A1C	A1C	5	-8	9	-8	7	**11**	1	3	-8	-1	-3	-8	10
2	A1C	A1L	3	-8	**11**	-8	7	9	1	5	-8	-1	-3	-8	10
3	A1C	A1R	3	-8	**11**	-8	7	9	1	5	-8	-1	-3	-8	10
4	A1C	A2C	3	-8	**11**	-8	9	7	1	5	-8	-1	-3	-8	10
5	A1C	A2L	5	-8	**11**	-8	7	9	1	3	-8	-1	-3	-8	10
6	A1C	A2R	5	-8	9	-8	7	**11**	1	3	-8	-1	-3	-8	10
7	A1C	A3C	5	-8	9	-8	7	**11**	1	3	-8	-1	-3	-8	10
8	A1C	A3L	3	-8	**11**	-8	7	9	-1	5	-8	1	-3	-8	10
9	A1C	A3R	5	-8	9	-8	7	**11**	1	3	-8	-1	-3	-8	10
10	A1C	C1C	5	-9	9	-9	7	**11**	-1	3	-9	-3	-5	1	40
11	A1C	C1L	3	-8	**11**	-8	7	9	1	5	-8	-3	-1	-8	40
12	A1C	C1R	3	-10	**11**	-7	7	9	-1	5	-10	-5	-3	1	40
13	A1C	C2C	3	-9	**11**	-9	7	9	1	5	-9	-3	-1	-5	40
14	A1C	C2L	3	-9	**11**	-9	7	9	-1	5	-9	-5	-3	1	40
15	A1C	C2R	3	-9	9	-9	7	**11**	-1	5	-9	-5	-3	1	40
16	A1C	C3C	5	-9	9	-9	7	**11**	1	3	-9	-3	-1	-5	40
17	A1C	C3L	3	-8	**11**	-8	7	9	1	5	-8	-3	-1	-8	40
18	A1C	C3R	3	-9	9	-9	7	**11**	1	5	-9	-3	-1	-5	40
19	A1C	P1C	5	-10	9	1	7	**11**	-3	3	-10	-7	-5	-1	50
20	A1C	P1L	3	-10	9	1	**11**	7	-3	5	-10	-7	-5	-1	50
21	A1C	P1R	3	-10	9	-1	**11**	7	-5	5	-10	-7	-3	1	50
22	A1C	P2C	5	-10	**11**	-1	9	7	-3	3	-10	-7	-5	1	50
23	A1C	P2L	5	-10	**11**	-7	7	9	-1	3	-10	-5	-3	1	50
24	A1C	P2R	5	-10	**11**	-1	7	9	-5	3	-10	-7	-3	1	50
25	A1C	P3C	3	-10	7	1	**11**	9	-3	5	-10	-7	-5	-1	50
26	A1C	P3L	3	-10	9	1	**11**	7	-3	5	-10	-7	-5	-1	50
27	A1C	P3R	3	-10	9	-1	**11**	7	-3	5	-10	-7	-5	1	50
28	A1C	STA	3	-8	**11**	-8	7	9	1	5	-8	-1	-3	-8	10
29	A1L	A1C	5	1	9	-11	-7	**11**	3	7	-1	-3	-5	-9	10
30	A1L	A1L	5	-1	9	-11	-7	**11**	3	7	1	-5	-3	-9	10
31	A1L	A1R	5	-1	9	-11	-7	**11**	3	7	1	-3	-5	-9	10
32	A1L	A2C	5	1	9	-11	-7	**11**	3	7	-1	-3	-5	-9	10
33	A1L	A2L	5	1	9	-11	-7	**11**	3	7	-1	-3	-5	-9	10

Table A.1: Algorithm wins and losses (continued)

	Objective Space Behaviour	Constraint Space Behaviour	RIGA	HyperM	DDECV	CCSaQPSO	SaQPSO	CCSaDE	SaDE	CCRIGA	CCHyperM	CCGVPSO	GVPSO	CCSaQGVPSO	Problem Changes
34	A1L	A2R	5	1	9	-11	-7	**11**	3	7	-1	-3	-5	-9	10
35	A1L	A3C	5	1	9	-11	-7	**11**	3	7	-1	-3	-5	-9	10
36	A1L	A3L	5	-1	9	-11	-7	**11**	3	7	1	-3	-5	-9	10
37	A1L	A3R	5	1	9	-11	-7	**11**	3	7	-1	-3	-5	-9	10
38	A1L	C1C	5	3	9	-11	-1	**11**	-3	7	1	-5	-9	-7	40
39	A1L	C1L	5	1	9	-11	-1	**11**	-3	7	3	-5	-9	-7	40
40	A1L	C1R	5	1	9	-11	-1	**11**	-3	7	3	-5	-9	-7	40
41	A1L	C2C	5	1	9	-9	-1	**11**	-3	7	3	-5	-11	-7	40
42	A1L	C2L	5	3	9	-11	-1	**11**	-3	7	1	-5	-9	-7	40
43	A1L	C2R	5	3	9	-11	-1	**11**	-3	7	1	-5	-9	-7	40
44	A1L	C3C	5	3	9	-9	-1	**11**	-3	7	1	-5	-11	-7	40
45	A1L	C3L	5	1	9	-11	-1	**11**	-3	7	3	-5	-9	-7	40
46	A1L	C3R	5	3	9	-11	-1	**11**	-3	7	1	-5	-7	-9	40
47	A1L	P1C	5	3	9	-7	-1	**11**	-3	7	1	-9	-11	-5	50
48	A1L	P1L	5	3	9	-5	-1	**11**	-3	7	1	-9	-11	-7	50
49	A1L	P1R	5	3	9	-5	-1	**11**	-3	7	1	-11	-9	-7	50
50	A1L	P2C	5	3	9	-7	-1	**11**	-3	7	1	-9	-11	-5	50
51	A1L	P2L	5	3	9	-7	-1	**11**	-3	7	1	-9	-11	-5	50
52	A1L	P2R	5	3	9	-5	-1	**11**	-3	7	1	-9	-11	-7	50
53	A1L	P3C	5	3	9	-5	-1	**11**	-3	7	1	-11	-9	-7	50
54	A1L	P3L	5	3	9	-5	-1	**11**	-3	7	1	-9	-11	-7	50
55	A1L	P3R	7	3	9	-5	-1	**11**	-3	5	1	-11	-9	-7	50
56	A1L	STA	5	1	9	-11	-7	**11**	3	7	-1	-3	-5	-9	10
57	A1R	A1C	7	1	9	-11	-7	**11**	-1	5	3	-3	-5	-9	10
58	A1R	A1L	7	-1	9	-9	-7	**11**	1	5	3	-3	-5	-11	10
59	A1R	A1R	7	-1	9	-11	-7	**11**	1	5	3	-3	-5	-9	10
60	A1R	A2C	7	-1	9	-11	-7	**11**	1	5	3	-3	-5	-9	10
61	A1R	A2L	5	1	9	-11	-7	**11**	-1	7	3	-5	-3	-9	10
62	A1R	A2R	7	1	9	-11	-7	**11**	-1	5	3	-5	-3	-9	10
63	A1R	A3C	7	1	9	-11	-7	**11**	-1	5	3	-3	-5	-9	10
64	A1R	A3L	7	-1	9	-11	-7	**11**	1	5	3	-3	-5	-9	10
65	A1R	A3R	7	3	9	-11	-7	**11**	-1	5	1	-3	-5	-9	10
66	A1R	C1C	7	1	9	-11	-1	**11**	-3	5	3	-5	-7	-9	40
67	A1R	C1L	7	1	9	-11	-1	**11**	-3	5	3	-5	-7	-9	40
68	A1R	C1R	7	1	9	-11	-1	**11**	-3	5	3	-5	-7	-9	40
69	A1R	C2C	5	1	9	-11	-1	**11**	-3	7	3	-5	-7	-9	40
70	A1R	C2L	5	1	9	-11	-1	**11**	-3	7	3	-5	-7	-9	40
71	A1R	C2R	7	1	9	-11	-1	**11**	-3	5	3	-5	-7	-9	40
72	A1R	C3C	7	1	9	-11	-1	**11**	-3	5	3	-5	-7	-9	40
73	A1R	C3L	7	1	9	-11	-1	**11**	-3	5	3	-5	-7	-9	40
74	A1R	C3R	7	3	9	-11	-1	**11**	-3	5	1	-5	-7	-9	40
75	A1R	P1C	7	1	9	-11	-1	**11**	-3	5	3	-5	-7	-9	50
76	A1R	P1L	7	1	9	-11	-1	**11**	-3	5	3	-5	-9	-7	50
77	A1R	P1R	7	1	9	-11	-1	**11**	-3	5	3	-7	-9	-5	50
78	A1R	P2C	7	1	9	-9	-1	**11**	-3	5	3	-5	-7	-11	50
79	A1R	P2L	7	1	9	-11	-1	**11**	-3	5	3	-7	-9	-5	50
80	A1R	P2R	7	1	9	-11	-1	**11**	-3	5	3	-5	-7	-9	50
81	A1R	P3C	7	1	9	-11	-1	**11**	-3	5	3	-7	-9	-5	50
82	A1R	P3L	7	1	9	-11	-1	**11**	-3	5	3	-7	-9	-5	50
83	A1R	P3R	7	1	9	-11	-1	**11**	-3	5	3	-7	-9	-5	50
84	A1R	STA	7	-1	9	-11	-7	**11**	1	5	3	-3	-5	-9	10
85	A2C	A1C	1	-3	9	-1	7	**11**	-7	3	-5	-11	-9	5	10
86	A2C	A1L	3	-3	9	-1	7	**11**	-5	5	-7	-9	-11	1	10
87	A2C	A1R	1	-3	9	-1	7	**11**	-5	5	-7	-11	-9	3	10
88	A2C	A2C	1	-5	9	-1	7	**11**	-7	5	-3	-11	-9	3	10
89	A2C	A2L	1	-7	9	-1	7	**11**	-5	5	-3	-11	-9	3	10

Table A.1: Algorithm wins and losses (continued)

	Objective Space Behaviour	Constraint Space Behaviour	RIGA	HyperM	DDECV	CCSaQPSO	SaQPSO	CCSaDE	SaDE	CCRIGA	CCHyperM	CCGVPSO	GVPSO	CCSaQGVPSO	Problem Changes
90	A2C	A2R	1	-5	9	-1	7	**11**	-7	5	-3	-11	-9	3	10
91	A2C	A3C	1	-3	9	-1	7	**11**	-7	3	-5	-11	-9	5	10
92	A2C	A3L	3	-5	9	-1	7	**11**	-3	5	-7	-9	-11	1	10
93	A2C	A3R	1	-5	9	-1	7	**11**	-3	3	-7	-11	-9	5	10
94	A2C	C1C	1	-3	9	-1	7	**11**	-7	3	-5	-11	-9	5	40
95	A2C	C1L	-1	-3	9	1	7	**11**	-7	5	-5	-11	-9	3	40
96	A2C	C1R	1	-3	9	-1	7	**11**	-7	5	-5	-11	-9	3	40
97	A2C	C2C	3	-5	9	-1	7	**11**	-7	5	-3	-11	-9	1	40
98	A2C	C2L	1	-5	9	-1	7	**11**	-7	3	-3	-9	-11	5	40
99	A2C	C2R	3	-5	9	-1	7	**11**	-7	5	-3	-11	-9	1	40
100	A2C	C3C	1	-3	9	-1	7	**11**	-7	3	-5	-11	-9	5	40
101	A2C	C3L	-1	-3	9	1	7	**11**	-7	5	-5	-11	-9	3	40
102	A2C	C3R	1	-3	9	-1	7	**11**	-7	5	-5	-11	-9	3	40
103	A2C	P1C	1	-3	9	-1	7	**11**	-7	3	-5	-11	-9	5	50
104	A2C	P1L	1	-3	9	-1	7	**11**	-7	5	-5	-9	-11	3	50
105	A2C	P1R	1	-3	9	-1	7	**11**	-7	5	-5	-9	-11	3	50
106	A2C	P2C	-1	-3	9	1	7	**11**	-7	5	-5	-11	-9	3	50
107	A2C	P2L	-1	-3	9	1	7	**11**	-7	5	-5	-11	-9	3	50
108	A2C	P2R	-1	-3	9	3	7	**11**	-7	1	-5	-11	-9	5	50
109	A2C	P3C	1	-3	9	-1	7	**11**	-7	5	-5	-9	-11	3	50
110	A2C	P3L	1	-3	9	-1	7	**11**	-7	3	-5	-9	-11	5	50
111	A2C	P3R	1	-3	9	-1	7	**11**	-7	5	-5	-11	-9	3	50
112	A2C	STA	1	-3	9	-1	7	**11**	-5	5	-7	-11	-9	3	10
113	A2L	A1C	1	-5	5	-1	9	7	-3	3	-7	-11	-9	**11**	10
114	A2L	A1L	3	-7	7	-1	5	9	-3	1	-5	-9	-11	**11**	10
115	A2L	A1R	3	-7	5	-1	9	7	-5	1	-3	-9	-11	**11**	10
116	A2L	A2C	3	-5	5	-1	7	9	-3	1	-7	-9	-11	**11**	10
117	A2L	A2L	1	-7	7	-1	5	**11**	-3	3	-5	-9	-11	9	10
118	A2L	A2R	3	-5	5	-1	7	9	-3	1	-7	-11	-9	**11**	10
119	A2L	A3C	1	-5	5	-1	9	7	-3	3	-7	-11	-9	**11**	10
120	A2L	A3L	3	-7	5	-1	9	7	-3	1	-5	-9	-11	**11**	10
121	A2L	A3R	1	-7	5	-1	9	7	-5	3	-3	-9	-11	**11**	10
122	A2L	C1C	1	-3	5	-1	9	7	-7	3	-5	-11	-9	**11**	40
123	A2L	C1L	1	-5	7	-1	5	9	-7	3	-3	-11	-9	**11**	40
124	A2L	C1R	3	-5	5	-1	7	9	-7	1	-3	-11	-9	**11**	40
125	A2L	C2C	3	-3	5	-1	7	9	-7	1	-5	-11	-9	**11**	40
126	A2L	C2L	1	-5	9	-1	5	**11**	-7	3	-3	-9	-11	7	40
127	A2L	C2R	1	-3	5	-1	7	9	-7	3	-5	-9	-11	**11**	40
128	A2L	C3C	1	-3	5	-1	9	7	-7	3	-5	-11	-9	**11**	40
129	A2L	C3L	1	-5	7	-1	5	9	-7	3	-3	-11	-9	**11**	40
130	A2L	C3R	1	-5	7	-1	5	9	-7	3	-3	-11	-9	**11**	40
131	A2L	P1C	1	-3	5	-1	9	7	-7	3	-5	-9	-11	**11**	50
132	A2L	P1L	1	-5	5	-1	7	9	-7	3	-3	-9	-11	**11**	50
133	A2L	P1R	1	-5	5	-1	7	9	-7	3	-3	-9	-11	**11**	50
134	A2L	P2C	1	-3	5	-1	9	7	-7	3	-5	-9	-11	**11**	50
135	A2L	P2L	1	-5	5	-1	9	7	-7	3	-3	-9	-11	**11**	50
136	A2L	P2R	1	-3	5	-1	9	7	-7	3	-5	-9	-11	**11**	50
137	A2L	P3C	3	-5	7	1	5	9	-7	-1	-3	-9	-11	**11**	50
138	A2L	P3L	1	-5	5	-1	7	9	-7	3	-3	-9	-11	**11**	50
139	A2L	P3R	3	-5	5	1	9	7	-7	-1	-3	-9	-11	**11**	50
140	A2L	STA	3	-7	5	-1	9	7	-3	1	-5	-9	-11	**11**	10
141	A2R	A1C	3	-3	**11**	-1	7	9	-5	5	-7	-9	-11	1	10
142	A2R	A1L	3	-5	**11**	-1	7	9	-3	5	-7	-11	-9	1	10
143	A2R	A1R	3	-5	**11**	-1	7	9	-3	5	-7	-9	-11	1	10
144	A2R	A2C	3	-7	**11**	-1	7	9	-3	5	-5	-9	-11	1	10
145	A2R	A2L	3	-11	**11**	-1	7	9	-3	5	-5	-7	-9	1	10

Table A.1: Algorithm wins and losses (continued)

	Objective Space Behaviour	Constraint Space Behaviour	RIGA	HyperM	DDECV	CCSaQPSO	SaQPSO	CCSaDE	SaDE	CCRIGA	CCHyperM	CCGVPSO	GVPSO	CCSaQGVPSO	Problem Changes
146	A2R	A2R	3	-7	**11**	-1	7	9	-3	5	-5	-11	-9	1	10
147	A2R	A3C	3	-3	**11**	-1	7	9	-5	5	-7	-9	-11	1	10
148	A2R	A3L	3	-7	**11**	-1	7	9	-3	5	-5	-11	-9	1	10
149	A2R	A3R	3	-7	9	-1	7	**11**	-3	5	-5	-9	-11	1	10
150	A2R	C1C	3	-3	**11**	-1	7	9	-7	5	-5	-9	-11	1	40
151	A2R	C1L	3	-3	**11**	-1	5	9	-7	7	-5	-9	-11	1	40
152	A2R	C1R	3	-3	9	-1	7	**11**	-7	5	-5	-11	-9	1	40
153	A2R	C2C	1	-5	**11**	-1	7	9	-7	5	-3	-11	-9	3	40
154	A2R	C2L	3	-5	**11**	-1	7	9	-7	5	-3	-9	-11	1	40
155	A2R	C2R	3	-5	**11**	-1	7	9	-7	5	-3	-11	-9	1	40
156	A2R	C3C	3	-3	**11**	-1	7	9	-7	5	-5	-9	-11	1	40
157	A2R	C3L	3	-5	**11**	-1	7	9	-7	5	-3	-9	-11	1	40
158	A2R	C3R	3	-5	9	-1	7	**11**	-7	5	-3	-9	-11	1	40
159	A2R	P1C	1	-3	**11**	-1	7	9	-7	3	-5	-9	-11	5	50
160	A2R	P1L	3	-3	**11**	-1	7	9	-7	5	-5	-11	-9	1	50
161	A2R	P1R	3	-3	**11**	-1	7	9	-7	5	-5	-11	-9	1	50
162	A2R	P2C	5	-3	**11**	-1	7	9	-7	1	-5	-9	-11	3	50
163	A2R	P2L	1	-3	**11**	-1	7	9	-7	3	-5	-9	-11	5	50
164	A2R	P2R	5	-3	**11**	-1	7	9	-7	1	-5	-9	-11	3	50
165	A2R	P3C	3	-3	**11**	-1	7	9	-7	5	-5	-9	-11	1	50
166	A2R	P3L	3	-3	**11**	-1	7	9	-7	5	-5	-9	-11	1	50
167	A2R	P3R	3	-3	**11**	-1	7	9	-7	5	-5	-9	-11	1	50
168	A2R	STA	3	-3	**11**	-1	7	9	-5	5	-7	-9	-11	1	10
169	A3C	A1C	5	-8	9	-8	7	**11**	1	3	-8	-3	-1	-8	10
170	A3C	A1L	3	-8	9	-8	7	**11**	1	5	-8	-3	-1	-8	10
171	A3C	A1R	5	-8	9	-8	7	**11**	1	3	-8	-1	-3	-8	10
172	A3C	A2C	5	-8	9	-8	7	**11**	1	3	-8	-3	-1	-8	10
173	A3C	A2L	3	-8	9	-8	7	**11**	1	5	-8	-1	-3	-8	10
174	A3C	A2R	5	-8	9	-8	7	**11**	1	3	-8	-3	-1	-8	10
175	A3C	A3C	5	-8	9	-8	7	**11**	1	3	-8	-3	-1	-8	10
176	A3C	A3L	3	-8	9	-8	7	**11**	1	5	-8	-3	-1	-8	10
177	A3C	A3R	5	-8	9	-8	7	**11**	1	3	-8	-3	-1	-8	10
178	A3C	C1C	5	-10	9	1	7	**11**	-3	3	-10	-7	-5	-1	40
179	A3C	C1L	3	-9	9	-9	7	**11**	1	5	-9	-5	-3	-1	40
180	A3C	C1R	3	-10	9	-1	7	**11**	-3	5	-10	-7	-5	1	40
181	A3C	C2C	5	-9	9	-9	7	**11**	-1	3	-9	-5	-3	1	40
182	A3C	C2L	3	-9	9	-9	7	**11**	-1	5	-9	-5	-3	1	40
183	A3C	C2R	5	-9	9	-9	7	**11**	-1	3	-9	-5	-3	1	40
184	A3C	C3C	3	-9	9	-9	7	**11**	-1	5	-9	-5	-3	1	40
185	A3C	C3L	3	-9	9	-5	7	**11**	1	5	-9	-3	-1	-9	40
186	A3C	C3R	5	-9	9	-9	7	**11**	-1	3	-9	-5	-3	1	40
187	A3C	P1C	5	-10	9	1	7	**11**	-3	3	-10	-5	-7	-1	50
188	A3C	P1L	5	-10	9	1	7	**11**	-1	3	-10	-5	-7	-3	50
189	A3C	P1R	5	-10	9	1	7	**11**	-3	3	-10	-5	-7	-1	50
190	A3C	P2C	5	-10	9	1	7	**11**	-3	3	-10	-5	-7	-1	50
191	A3C	P2L	5	-10	9	-3	7	**11**	-1	3	-10	-7	-5	1	50
192	A3C	P2R	5	-10	9	1	7	**11**	-3	3	-10	-5	-7	-1	50
193	A3C	P3C	5	-10	9	1	7	**11**	-3	3	-10	-5	-7	-1	50
194	A3C	P3L	5	-10	9	1	7	**11**	-1	3	-10	-3	-5	-7	50
195	A3C	P3R	5	-10	9	1	7	**11**	-3	3	-10	-5	-7	-1	50
196	A3C	STA	5	-8	9	-8	7	**11**	1	3	-8	-3	-1	-8	10
197	A3L	A1C	7	3	9	-11	-7	**11**	-1	5	1	-3	-5	-9	10
198	A3L	A1L	7	1	9	-11	-7	**11**	-1	5	3	-5	-3	-9	10
199	A3L	A1R	7	1	9	-11	-7	**11**	-1	5	3	-5	-3	-9	10
200	A3L	A2C	7	1	9	-11	-7	**11**	-1	5	3	-3	-5	-9	10
201	A3L	A2L	5	3	9	-11	-7	**11**	-1	7	1	-5	-3	-9	10

Table A.1: Algorithm wins and losses (continued)

	Objective Space Behaviour	Constraint Space Behaviour	RIGA	HyperM	DDECV	CCSaQPSO	SaQPSO	CCSaDE	SaDE	CCRIGA	CCHyperM	CCGVPSO	GVPSO	CCSaQGVPSO	Problem Changes
202	A3L	A2R	5	3	9	-11	-7	**11**	-1	7	1	-3	-5	-9	10
203	A3L	A3C	7	3	9	-11	-7	**11**	-1	5	1	-3	-5	-9	10
204	A3L	A3L	7	3	9	-11	-7	**11**	-1	5	1	-3	-5	-9	10
205	A3L	A3R	7	1	9	-11	-7	**11**	-1	5	3	-5	-3	-9	10
206	A3L	C1C	7	3	9	-11	-1	**11**	-3	5	1	-5	-7	-9	40
207	A3L	C1L	5	1	9	-11	-1	**11**	-3	7	3	-7	-5	-9	40
208	A3L	C1R	5	1	9	-11	-1	**11**	-3	7	3	-5	-7	-9	40
209	A3L	C2C	7	3	9	-9	-1	**11**	-3	5	1	-5	-7	-11	40
210	A3L	C2L	5	3	9	-11	-1	**11**	-3	7	1	-5	-7	-9	40
211	A3L	C2R	7	3	9	-11	-1	**11**	-3	5	1	-5	-7	-9	40
212	A3L	C3C	7	3	9	-11	-1	**11**	-3	5	1	-5	-7	-9	40
213	A3L	C3L	5	3	9	-11	-1	**11**	-3	7	1	-5	-7	-9	40
214	A3L	C3R	5	1	9	-11	-1	**11**	-3	7	3	-5	-7	-9	40
215	A3L	P1C	7	3	9	-11	-1	**11**	-3	5	1	-7	-9	-5	50
216	A3L	P1L	7	3	9	-5	-1	**11**	-3	5	1	-9	-11	-7	50
217	A3L	P1R	7	3	9	-5	-1	**11**	-3	5	1	-9	-11	-7	50
218	A3L	P2C	7	3	9	-9	-1	**11**	-3	5	1	-7	-11	-5	50
219	A3L	P2L	7	1	9	-9	-1	**11**	-3	5	3	-7	-11	-5	50
220	A3L	P2R	7	1	9	-7	-1	**11**	-3	5	3	-9	-11	-5	50
221	A3L	P3C	7	3	9	-7	-1	**11**	-3	5	1	-9	-11	-5	50
222	A3L	P3L	7	3	9	-7	-1	**11**	-3	5	1	-9	-11	-5	50
223	A3L	P3R	7	3	9	-5	-1	**11**	-3	5	1	-9	-11	-7	50
224	A3L	STA	7	3	9	-11	-7	**11**	-1	5	1	-5	-3	-9	10
225	A3R	A1C	7	1	9	-11	-7	**11**	-1	5	3	-5	-3	-9	10
226	A3R	A1L	7	1	9	-11	-7	**11**	-1	5	3	-5	-3	-9	10
227	A3R	A1R	7	1	9	-11	-7	**11**	-1	5	3	-3	-5	-9	10
228	A3R	A2C	7	1	9	-11	-7	**11**	-1	5	3	-3	-5	-9	10
229	A3R	A2L	7	1	**11**	-11	-7	9	-1	5	3	-3	-5	-9	10
230	A3R	A2R	7	1	9	-11	-7	**11**	-1	5	3	-3	-5	-9	10
231	A3R	A3C	7	1	9	-11	-7	**11**	-1	5	3	-3	-5	-9	10
232	A3R	A3L	7	1	9	-11	-7	**11**	-1	5	3	-3	-5	-9	10
233	A3R	A3R	7	1	9	-11	-7	**11**	-1	5	3	-5	-3	-9	10
234	A3R	C1C	7	1	9	-11	-3	**11**	-1	5	3	-7	-5	-9	40
235	A3R	C1L	5	1	9	-11	-1	**11**	-3	7	3	-7	-5	-9	40
236	A3R	C1R	7	1	9	-11	-3	**11**	-1	5	3	-7	-5	-9	40
237	A3R	C2C	5	1	9	-11	-3	**11**	-1	7	3	-5	-7	-9	40
238	A3R	C2L	5	1	9	-11	-3	**11**	-1	7	3	-5	-7	-9	40
239	A3R	C2R	7	1	9	-11	-3	**11**	-1	5	3	-5	-7	-9	40
240	A3R	C3C	7	1	9	-11	-3	**11**	-1	5	3	-7	-5	-9	40
241	A3R	C3L	7	1	9	-11	-1	**11**	-3	5	3	-7	-5	-9	40
242	A3R	C3R	7	1	9	-11	-3	**11**	-1	5	3	-5	-7	-9	40
243	A3R	P1C	7	1	9	-11	-1	**11**	-3	5	3	-7	-9	-5	50
244	A3R	P1L	7	1	9	-11	-1	**11**	-3	5	3	-7	-9	-5	50
245	A3R	P1R	7	1	9	-11	-1	**11**	-3	5	3	-9	-7	-5	50
246	A3R	P2C	7	1	9	-9	-1	**11**	-3	5	3	-5	-7	-11	50
247	A3R	P2L	7	1	9	-11	-1	**11**	-3	5	3	-5	-7	-9	50
248	A3R	P2R	7	1	9	-11	-1	**11**	-3	5	3	-5	-7	-9	50
249	A3R	P3C	7	1	9	-11	-1	**11**	-3	5	3	-7	-9	-5	50
250	A3R	P3L	7	1	9	-11	-1	**11**	-3	5	3	-5	-7	-9	50
251	A3R	P3R	7	1	9	-9	-1	**11**	-3	5	3	-5	-7	-11	50
252	A3R	STA	7	1	9	-11	-7	**11**	-1	5	3	-3	-5	-9	10
253	C1C	A1C	7	-9	9	-9	3	**11**	1	5	-9	-3	-1	-5	50
254	C1C	A1L	7	-10	**11**	-7	5	9	-3	3	-10	-5	-1	1	50
255	C1C	A1R	7	-10	**11**	1	3	9	-5	5	-10	-7	-3	-1	50
256	C1C	A2C	5	-9	**11**	-9	3	9	1	7	-9	-3	-1	-5	50
257	C1C	A2L	5	-9	**11**	-9	3	9	-3	7	-9	-5	-1	1	50

Table A.1: Algorithm wins and losses (continued)

	Objective Space Behaviour	Constraint Space Behaviour	RIGA	HyperM	DDECV	CCSaQPSO	SaQPSO	CCSaDE	SaDE	CCRIGA	CCHyperM	CCGVPSO	GVPSO	CCSaQGVPSO	Problem Changes
258	C1C	A2R	5	-9	9	-9	3	**11**	-3	7	-9	-5	-1	1	50
259	C1C	A3C	7	-10	9	1	3	**11**	-1	5	-10	-5	-3	-7	50
260	C1C	A3L	5	-10	**11**	-7	3	9	-1	7	-10	-3	1	-5	50
261	C1C	A3R	7	-9	**11**	-9	3	9	-3	5	-9	-5	-1	1	50
262	C1C	C1C	7	-10	9	3	-7	**11**	-1	5	-10	-5	-3	1	67
263	C1C	C1L	5	-8	**11**	-8	3	9	1	7	-8	-3	-1	-8	67
264	C1C	C1R	7	-10	**11**	3	-7	9	-1	5	-10	-5	-3	1	67
265	C1C	C2C	5	-10	9	1	3	**11**	-1	7	-10	-5	-3	-7	67
266	C1C	C2L	5	-10	**11**	-7	3	9	-1	7	-10	-3	-5	1	67
267	C1C	C2R	5	-10	9	1	3	**11**	-3	7	-10	-7	-5	-1	67
268	C1C	C3C	7	-10	9	3	-7	**11**	-1	5	-10	-5	-3	1	67
269	C1C	C3L	5	-7	**11**	-7	-7	9	3	7	-7	1	-1	-7	67
270	C1C	C3R	7	-10	**11**	-1	3	9	-3	5	-10	-7	-5	1	67
271	C1C	P1C	7	-8	9	-8	3	**11**	1	5	-8	-3	-1	-8	50
272	C1C	P1L	3	-10	**11**	1	7	9	-3	5	-10	-5	-1	-7	50
273	C1C	P1R	5	-10	**11**	1	3	9	-3	7	-10	-5	-1	-7	50
274	C1C	P2C	7	-10	9	-7	5	**11**	-3	3	-10	-5	-1	1	50
275	C1C	P2L	7	-9	9	-9	3	**11**	-3	5	-9	-5	-1	1	50
276	C1C	P2R	3	-10	9	-1	5	**11**	-3	7	-10	-7	-5	1	50
277	C1C	P3C	5	-10	**11**	1	3	9	-5	7	-10	-7	-3	-1	50
278	C1C	P3L	5	-9	**11**	-5	3	9	-1	7	-9	-3	1	-9	50
279	C1C	P3R	5	-10	**11**	-5	3	9	-1	7	-10	-3	1	-7	50
280	C1C	STA	5	-8	**11**	-8	3	9	-1	7	-8	-3	1	-8	50
281	C1L	A1C	5	1	9	-11	-7	**11**	-1	7	3	-5	-3	-9	50
282	C1L	A1L	5	1	**11**	-9	-7	9	-1	7	3	-3	-5	-11	50
283	C1L	A1R	5	1	9	-11	-7	**11**	-1	7	3	-3	-5	-9	50
284	C1L	A2C	5	1	**11**	-11	-7	9	-1	7	3	-5	-3	-9	50
285	C1L	A2L	5	1	9	-9	-7	**11**	-1	7	3	-3	-5	-11	50
286	C1L	A2R	5	1	**11**	-11	-7	9	-1	7	3	-5	-3	-9	50
287	C1L	A3C	7	1	9	-11	-7	**11**	-1	5	3	-5	-3	-9	50
288	C1L	A3L	7	1	9	-11	-7	**11**	-1	5	3	-3	-5	-9	50
289	C1L	A3R	5	1	9	-11	-7	**11**	-1	7	3	-5	-3	-9	50
290	C1L	C1C	5	1	9	-9	-7	**11**	-1	7	3	-5	-3	-11	67
291	C1L	C1L	5	1	**11**	-9	-5	9	-1	7	3	-7	-3	-11	67
292	C1L	C1R	5	1	9	-9	-5	**11**	-1	7	3	-7	-3	-11	67
293	C1L	C2C	5	1	9	-9	-7	**11**	-1	7	3	-5	-3	-11	67
294	C1L	C2L	5	1	9	-9	-7	**11**	-1	7	3	-3	-5	-11	67
295	C1L	C2R	5	1	9	-9	-7	**11**	-1	7	3	-3	-5	-11	67
296	C1L	C3C	7	1	9	-9	-7	**11**	-1	5	3	-5	-3	-11	67
297	C1L	C3L	5	1	9	-11	-7	**11**	-1	7	3	-5	-3	-9	67
298	C1L	C3R	5	1	9	-9	-7	**11**	-1	7	3	-5	-3	-11	67
299	C1L	P1C	5	1	9	-11	-7	**11**	-1	7	3	-5	-3	-9	50
300	C1L	P1L	5	1	9	-9	-7	**11**	-1	7	3	-5	-3	-11	50
301	C1L	P1R	5	1	9	-11	-7	**11**	-1	7	3	-5	-3	-9	50
302	C1L	P2C	5	1	9	-11	-7	**11**	-1	7	3	-3	-5	-9	50
303	C1L	P2L	5	1	9	-11	-7	**11**	-1	7	3	-5	-3	-9	50
304	C1L	P2R	5	1	9	-11	-7	**11**	-1	7	3	-5	-3	-9	50
305	C1L	P3C	5	1	9	-9	-7	**11**	-1	7	3	-5	-3	-11	50
306	C1L	P3L	5	1	9	-9	-7	**11**	-1	7	3	-5	-3	-11	50
307	C1L	P3R	3	1	9	-11	-7	**11**	-1	7	5	-5	-3	-9	50
308	C1L	STA	5	1	9	-11	-7	**11**	-1	7	3	-3	-5	-9	50
309	C1R	A1C	5	3	9	-11	-5	**11**	-1	1	7	-3	-7	-9	50
310	C1R	A1L	5	3	9	-11	-7	**11**	-1	1	7	-3	-5	-9	50
311	C1R	A1R	5	3	9	-11	-7	**11**	-1	1	7	-3	-5	-9	50
312	C1R	A2C	5	1	9	-11	-7	**11**	-1	3	7	-3	-5	-9	50
313	C1R	A2L	3	5	9	-11	-3	**11**	-1	1	7	-7	-5	-9	50

Table A.1: Algorithm wins and losses (continued)

	Objective Space Behaviour	Constraint Space Behaviour	RIGA	HyperM	DDECV	CCSaQPSO	SaQPSO	CCSaDE	SaDE	CCRIGA	CCHyperM	CCGVPSO	GVPSO	CCSaQGVPSO	Problem Changes
314	C1R	A2R	5	1	9	-11	-5	**11**	-1	3	7	-3	-7	-9	50
315	C1R	A3C	5	3	9	-11	-7	**11**	-1	1	7	-3	-5	-9	50
316	C1R	A3L	5	3	9	-11	-7	**11**	-1	1	7	-3	-5	-9	50
317	C1R	A3R	5	3	9	-9	-5	**11**	-1	1	7	-3	-7	-11	50
318	C1R	C1C	5	3	9	-9	-3	**11**	-1	1	7	-7	-5	-11	67
319	C1R	C1L	5	3	9	-9	-3	**11**	-1	1	7	-7	-5	-11	67
320	C1R	C1R	5	3	9	-11	-3	**11**	-1	1	7	-5	-7	-9	67
321	C1R	C2C	5	1	9	-9	-3	**11**	-1	3	7	-7	-5	-11	67
322	C1R	C2L	3	5	9	-11	-3	**11**	-1	1	7	-7	-5	-9	67
323	C1R	C2R	5	1	9	-9	-3	**11**	-1	3	7	-7	-5	-11	67
324	C1R	C3C	5	3	9	-9	-3	**11**	-1	1	7	-7	-5	-11	67
325	C1R	C3L	5	1	9	-9	-7	**11**	-1	3	7	-5	-3	-11	67
326	C1R	C3R	5	3	9	-11	-3	**11**	-1	1	7	-5	-7	-9	67
327	C1R	P1C	5	3	9	-11	-3	**11**	-1	1	7	-5	-7	-9	50
328	C1R	P1L	5	3	9	-11	-7	**11**	-1	1	7	-3	-5	-9	50
329	C1R	P1R	5	3	9	-11	-7	**11**	-1	1	7	-3	-5	-9	50
330	C1R	P2C	5	1	9	-9	-3	**11**	-1	3	7	-7	-5	-11	50
331	C1R	P2L	5	1	9	-11	-3	**11**	-1	3	7	-5	-7	-9	50
332	C1R	P2R	5	1	9	-11	-3	**11**	-1	3	7	-7	-5	-9	50
333	C1R	P3C	5	3	9	-11	-7	**11**	-1	1	7	-3	-5	-9	50
334	C1R	P3L	5	3	9	-11	-7	**11**	-1	1	7	-3	-5	-9	50
335	C1R	P3R	5	3	9	-11	-7	**11**	-1	1	7	-3	-5	-9	50
336	C1R	STA	5	3	9	-11	-7	**11**	-1	1	7	-3	-5	-9	50
337	C2C	A1C	-1	-3	9	3	7	**11**	-7	1	-5	-11	-9	5	50
338	C2C	A1L	-1	-3	9	3	7	**11**	-7	1	-5	-9	-11	5	50
339	C2C	A1R	-1	-5	9	3	7	**11**	-7	1	-3	-9	-11	5	50
340	C2C	A2C	-1	-5	9	3	7	**11**	-7	1	-3	-11	-9	5	50
341	C2C	A2L	-1	-5	9	3	7	**11**	-7	1	-3	-9	-11	5	50
342	C2C	A2R	-1	-5	9	3	7	**11**	-7	1	-3	-11	-9	5	50
343	C2C	A3C	-1	-5	9	3	7	**11**	-7	1	-3	-11	-9	5	50
344	C2C	A3L	-1	-5	9	3	7	**11**	-7	1	-3	-9	-11	5	50
345	C2C	A3R	-1	-5	9	3	7	**11**	-7	1	-3	-9	-11	5	50
346	C2C	C1C	-1	-3	9	3	7	**11**	-7	1	-5	-9	-11	5	67
347	C2C	C1L	-1	-3	9	3	7	**11**	-7	1	-5	-9	-11	5	67
348	C2C	C1R	-1	-5	9	3	7	**11**	-7	1	-3	-11	-9	5	67
349	C2C	C2C	-1	-5	9	3	7	**11**	-7	1	-3	-9	-11	5	67
350	C2C	C2L	-1	-5	9	3	7	**11**	-7	1	-3	-9	-11	5	67
351	C2C	C2R	-1	-5	9	3	7	**11**	-7	1	-3	-9	-11	5	67
352	C2C	C3C	-1	-5	9	3	7	**11**	-7	1	-3	-9	-11	5	67
353	C2C	C3L	-1	-3	9	3	7	**11**	-7	1	-5	-9	-11	5	67
354	C2C	C3R	-1	-5	9	3	7	**11**	-7	1	-3	-11	-9	5	67
355	C2C	P1C	-1	-3	9	3	7	**11**	-7	1	-5	-11	-9	5	50
356	C2C	P1L	-1	-3	9	3	7	**11**	-7	1	-5	-9	-11	5	50
357	C2C	P1R	-1	-3	9	3	7	**11**	-7	1	-5	-9	-11	5	50
358	C2C	P2C	-1	-3	9	3	7	**11**	-7	1	-5	-9	-11	5	50
359	C2C	P2L	-1	-3	9	3	7	**11**	-7	1	-5	-11	-9	5	50
360	C2C	P2R	-1	-3	9	3	7	**11**	-7	1	-5	-11	-9	5	50
361	C2C	P3C	-1	-3	9	3	7	**11**	-7	1	-5	-9	-11	5	50
362	C2C	P3L	-1	-3	9	3	7	**11**	-7	1	-5	-9	-11	5	50
363	C2C	P3R	-1	-3	9	3	7	**11**	-7	1	-5	-9	-11	5	50
364	C2C	STA	-1	-3	9	3	7	**11**	-7	1	-5	-9	-11	5	50
365	C2L	A1C	5	-5	-1	1	7	9	-7	3	-3	-9	-11	**11**	50
366	C2L	A1L	5	-5	1	3	7	9	-7	-1	-3	-9	-11	**11**	50
367	C2L	A1R	-1	-5	5	1	7	9	-7	3	-3	-9	-11	**11**	50
368	C2L	A2C	-1	-5	1	5	7	9	-7	3	-3	-9	-11	**11**	50
369	C2L	A2L	-1	-3	3	5	7	9	-7	1	-5	-9	-11	**11**	50

Table A.1: Algorithm wins and losses (continued)

	Objective Space Behaviour	Constraint Space Behaviour	RIGA	HyperM	DDECV	CCSaQPSO	SaQPSO	CCSaDE	SaDE	CCRIGA	CCHyperM	CCGVPSO	GVPSO	CCSaQGVPSO	Problem Changes
370	C2L	A2R	-1	-5	1	5	7	9	-7	3	-3	-9	-11	**11**	50
371	C2L	A3C	5	-5	-1	3	7	9	-7	1	-3	-9	-11	**11**	50
372	C2L	A3L	-1	-5	3	5	7	9	-7	1	-3	-9	-11	**11**	50
373	C2L	A3R	5	-5	1	3	7	9	-7	-1	-3	-9	-11	**11**	50
374	C2L	C1C	5	-5	-1	3	7	9	-7	1	-3	-11	-9	**11**	67
375	C2L	C1L	-1	-5	1	5	7	9	-7	3	-3	-11	-9	**11**	67
376	C2L	C1R	-1	-5	5	3	7	9	-7	1	-3	-9	-11	**11**	67
377	C2L	C2C	-1	-5	3	5	7	9	-7	1	-3	-11	-9	**11**	67
378	C2L	C2L	-1	-3	7	5	1	9	-7	3	-5	-11	-9	**11**	67
379	C2L	C2R	-1	-5	3	5	7	9	-7	1	-3	-11	-9	**11**	67
380	C2L	C3C	1	-5	-1	5	7	9	-7	3	-3	-9	-11	**11**	67
381	C2L	C3L	-1	-5	1	3	7	9	-7	5	-3	-9	-11	**11**	67
382	C2L	C3R	-1	-5	5	3	7	9	-7	1	-3	-11	-9	**11**	67
383	C2L	P1C	5	-5	-1	1	7	9	-7	3	-3	-9	-11	**11**	50
384	C2L	P1L	1	-5	-1	5	7	9	-7	3	-3	-9	-11	**11**	50
385	C2L	P1R	1	-5	-1	5	7	9	-7	3	-3	-9	-11	**11**	50
386	C2L	P2C	-1	-3	1	5	7	9	-7	3	-5	-9	-11	**11**	50
387	C2L	P2L	-1	-5	3	5	7	9	-7	1	-3	-9	-11	**11**	50
388	C2L	P2R	-1	-3	1	5	7	9	-7	3	-5	-9	-11	**11**	50
389	C2L	P3C	1	-5	-1	5	7	9	-7	3	-3	-9	-11	**11**	50
390	C2L	P3L	1	-5	-1	5	7	9	-7	3	-3	-9	-11	**11**	50
391	C2L	P3R	1	-5	-1	5	7	9	-7	3	-3	-9	-11	**11**	50
392	C2L	STA	-1	-5	1	5	7	9	-7	3	-3	-9	-11	**11**	50
393	C2R	A1C	-1	-3	9	3	7	**11**	-7	1	-5	-11	-9	5	50
394	C2R	A1L	-1	-3	9	1	7	**11**	-7	3	-5	-11	-9	5	50
395	C2R	A1R	-1	-5	9	1	7	**11**	-7	3	-3	-11	-9	5	50
396	C2R	A2C	-1	-5	9	3	7	**11**	-7	1	-3	-11	-9	5	50
397	C2R	A2L	-1	-5	9	3	7	**11**	-7	1	-3	-11	-9	5	50
398	C2R	A2R	-1	-5	9	3	7	**11**	-7	1	-3	-11	-9	5	50
399	C2R	A3C	-1	-3	9	3	7	**11**	-7	1	-5	-11	-9	5	50
400	C2R	A3L	-1	-5	9	3	7	**11**	-7	1	-3	-11	-9	5	50
401	C2R	A3R	-1	-5	9	1	7	**11**	-7	5	-3	-9	-11	3	50
402	C2R	C1C	-1	-3	9	5	7	**11**	-7	1	-5	-9	-11	3	67
403	C2R	C1L	-1	-3	9	3	7	**11**	-7	1	-5	-9	-11	5	67
404	C2R	C1R	-1	-5	9	1	7	**11**	-7	5	-3	-9	-11	3	67
405	C2R	C2C	-1	-5	9	3	7	**11**	-7	1	-3	-9	-11	5	67
406	C2R	C2L	-1	-5	9	5	7	**11**	-7	1	-3	-9	-11	3	67
407	C2R	C2R	-1	-5	9	5	7	**11**	-7	3	-3	-9	-11	1	67
408	C2R	C3C	-1	-3	9	5	7	**11**	-7	1	-5	-9	-11	3	67
409	C2R	C3L	-1	-5	9	3	7	**11**	-7	1	-3	-9	-11	5	67
410	C2R	C3R	-1	-5	9	1	7	**11**	-7	5	-3	-9	-11	3	67
411	C2R	P1C	-1	-3	9	3	7	**11**	-7	1	-5	-11	-9	5	50
412	C2R	P1L	-1	-3	9	3	7	**11**	-7	5	-5	-11	-9	1	50
413	C2R	P1R	-1	-3	9	3	7	**11**	-7	5	-5	-11	-9	1	50
414	C2R	P2C	-1	-3	9	3	7	**11**	-7	5	-5	-11	-9	1	50
415	C2R	P2L	-1	-3	9	1	7	**11**	-7	5	-5	-11	-9	3	50
416	C2R	P2R	-1	-3	9	3	7	**11**	-7	5	-5	-11	-9	1	50
417	C2R	P3C	-1	-5	9	3	7	**11**	-7	5	-3	-11	-9	1	50
418	C2R	P3L	-1	-3	9	3	7	**11**	-7	5	-5	-11	-9	1	50
419	C2R	P3R	-1	-3	9	3	7	**11**	-7	5	-5	-11	-9	1	50
420	C2R	STA	-1	-5	9	3	7	**11**	-7	5	-3	-11	-9	1	50
421	C3C	A1C	5	-10	9	-1	7	**11**	-3	3	-10	-7	-5	1	50
422	C3C	A1L	5	-10	9	1	7	**11**	-3	3	-10	-5	-7	-1	50
423	C3C	A1R	5	-10	9	1	7	**11**	-1	3	-10	-3	-5	-7	50
424	C3C	A2C	3	-10	9	-1	7	**11**	-3	5	-10	-5	-7	1	50
425	C3C	A2L	3	-10	9	1	7	**11**	-3	5	-10	-7	-5	-1	50

Table A.1: Algorithm wins and losses (continued)

	Objective Space Behaviour	Constraint Space Behaviour	RIGA	HyperM	DDECV	CCSaQPSO	SaQPSO	CCSaDE	SaDE	CCRIGA	CCHyperM	CCGVPSO	GVPSO	CCSaQGVPSO	Problem Changes
426	C3C	A2R	5	-10	9	-1	7	**11**	-3	3	-10	-7	-5	1	50
427	C3C	A3C	5	-10	9	-1	7	**11**	-3	3	-10	-7	-5	1	50
428	C3C	A3L	5	-10	9	-1	7	**11**	-3	3	-10	-5	-7	1	50
429	C3C	A3R	5	-10	9	-1	7	**11**	-3	3	-10	-5	-7	1	50
430	C3C	C1C	7	-10	9	1	3	**11**	-3	5	-10	-7	-5	-1	67
431	C3C	C1L	3	-9	9	-5	7	**11**	1	5	-9	-3	-1	-9	67
432	C3C	C1R	3	-10	9	-1	5	**11**	-3	7	-10	-7	-5	1	67
433	C3C	C2C	7	-10	9	1	3	**11**	-3	5	-10	-7	-5	-1	67
434	C3C	C2L	3	-10	9	1	5	**11**	-1	7	-10	-5	-3	-7	67
435	C3C	C2R	7	-10	9	1	3	**11**	-3	5	-10	-7	-5	-1	67
436	C3C	C3C	3	-10	9	-1	7	**11**	-3	5	-10	-5	-7	1	67
437	C3C	C3L	5	-10	9	1	3	**11**	-1	7	-10	-5	-3	-7	67
438	C3C	C3R	5	-10	9	-1	7	**11**	-3	3	-10	-7	-5	1	67
439	C3C	P1C	5	-10	9	-1	7	**11**	-3	3	-10	-7	-5	1	50
440	C3C	P1L	3	-10	9	-1	7	**11**	-3	5	-10	-5	-7	1	50
441	C3C	P1R	3	-10	9	-1	7	**11**	-3	5	-10	-5	-7	1	50
442	C3C	P2C	3	-10	9	-3	7	**11**	-1	5	-10	-7	-5	1	50
443	C3C	P2L	3	-10	9	-1	7	**11**	-3	5	-10	-7	-5	1	50
444	C3C	P2R	5	-10	9	-1	7	**11**	-3	3	-10	-7	-5	1	50
445	C3C	P3C	3	-10	9	-1	7	**11**	-3	5	-10	-7	-5	1	50
446	C3C	P3L	3	-10	9	-1	7	**11**	-3	5	-10	-5	-7	1	50
447	C3C	P3R	3	-10	9	-1	7	**11**	-3	5	-10	-7	-5	1	50
448	C3C	STA	3	-10	9	-1	7	**11**	-3	5	-10	-5	-7	1	50
449	C3L	A1C	3	7	9	-9	-7	**11**	-1	1	5	-5	-3	-11	50
450	C3L	A1L	1	7	9	-9	-7	**11**	-1	3	5	-3	-5	-11	50
451	C3L	A1R	1	7	9	-9	-7	**11**	-1	3	5	-3	-5	-11	50
452	C3L	A2C	1	7	9	-9	-7	**11**	-1	3	5	-3	-5	-11	50
453	C3L	A2L	1	7	9	-9	-7	**11**	-1	3	5	-3	-5	-11	50
454	C3L	A2R	1	7	9	-11	-7	**11**	-1	3	5	-3	-5	-9	50
455	C3L	A3C	3	7	9	-9	-7	**11**	-1	1	5	-5	-3	-11	50
456	C3L	A3L	1	7	9	-9	-7	**11**	-1	3	5	-3	-5	-11	50
457	C3L	A3R	1	7	9	-9	-7	**11**	-1	3	5	-3	-5	-11	50
458	C3L	C1C	3	7	9	-9	-7	**11**	-1	1	5	-3	-5	-11	67
459	C3L	C1L	1	5	9	-11	-7	**11**	-1	3	7	-3	-5	-9	67
460	C3L	C1R	1	7	9	-9	-7	**11**	-1	3	5	-3	-5	-11	67
461	C3L	C2C	1	7	9	-9	-7	**11**	-1	3	5	-3	-5	-11	67
462	C3L	C2L	1	7	9	-9	-7	**11**	-1	3	5	-3	-5	-11	67
463	C3L	C2R	1	7	9	-11	-7	**11**	-1	3	5	-3	-5	-9	67
464	C3L	C3C	3	7	9	-9	-7	**11**	-1	1	5	-3	-5	-11	67
465	C3L	C3L	3	7	9	-11	-7	**11**	-1	1	5	-5	-3	-9	67
466	C3L	C3R	1	7	9	-9	-7	**11**	-1	3	5	-3	-5	-11	67
467	C3L	P1C	3	7	9	-9	-7	**11**	-1	1	5	-5	-3	-11	50
468	C3L	P1L	1	7	9	-9	-7	**11**	-1	3	5	-5	-3	-11	50
469	C3L	P1R	1	7	9	-9	-7	**11**	-1	3	5	-5	-3	-11	50
470	C3L	P2C	1	7	9	-9	-7	**11**	-1	3	5	-3	-5	-11	50
471	C3L	P2L	1	7	9	-9	-7	**11**	-1	3	5	-3	-5	-11	50
472	C3L	P2R	1	7	9	-9	-7	**11**	-1	3	5	-3	-5	-11	50
473	C3L	P3C	1	7	9	-9	-7	**11**	-1	3	5	-3	-5	-11	50
474	C3L	P3L	1	7	9	-9	-7	**11**	-1	3	5	-3	-5	-11	50
475	C3L	P3R	1	7	9	-9	-7	**11**	-1	3	5	-3	-5	-11	50
476	C3L	STA	1	7	9	-9	-7	**11**	-1	3	5	-3	-5	-11	50
477	C3R	A1C	3	7	9	-11	-5	**11**	-1	1	5	-7	-3	-9	50
478	C3R	A1L	3	5	9	-11	-5	**11**	-1	1	7	-7	-3	-9	50
479	C3R	A1R	3	5	9	-11	-5	**11**	-1	1	7	-7	-3	-9	50
480	C3R	A2C	3	7	9	-11	-3	**11**	-1	1	5	-7	-5	-9	50
481	C3R	A2L	3	7	9	-11	-5	**11**	-1	1	5	-7	-3	-9	50

Table A.1: Algorithm wins and losses (continued)

	Objective Space Behaviour	Constraint Space Behaviour	RIGA	HyperM	DDECV	CCSaQPSO	SaQPSO	CCSaDE	SaDE	CCRIGA	CCHyperM	CCGVPSO	GVPSO	CCSaQGVPSO	Problem Changes
482	C3R	A2R	3	5	9	-11	-5	**11**	-1	1	7	-7	-3	-9	50
483	C3R	A3C	3	7	9	-11	-3	**11**	-1	1	5	-7	-5	-9	50
484	C3R	A3L	3	5	9	-11	-3	**11**	-1	1	7	-7	-5	-9	50
485	C3R	A3R	3	5	9	-11	-5	**11**	-1	1	7	-7	-3	-9	50
486	C3R	C1C	3	7	9	-11	-3	**11**	-1	1	5	-7	-5	-9	67
487	C3R	C1L	3	5	9	-11	-3	**11**	-1	1	7	-7	-5	-9	67
488	C3R	C1R	3	5	9	-11	-3	**11**	-1	1	7	-7	-5	-9	67
489	C3R	C2C	3	7	9	-9	-3	**11**	-1	1	5	-7	-5	-11	67
490	C3R	C2L	3	7	9	-9	-3	**11**	-1	1	5	-7	-5	-11	67
491	C3R	C2R	3	5	9	-11	-3	**11**	-1	1	7	-7	-5	-9	67
492	C3R	C3C	3	7	9	-11	-3	**11**	-1	1	5	-7	-5	-9	67
493	C3R	C3L	3	5	9	-11	-3	**11**	-1	1	7	-7	-5	-9	67
494	C3R	C3R	1	5	9	-11	-3	**11**	-1	3	7	-7	-5	-9	67
495	C3R	P1C	3	7	9	-11	-5	**11**	-1	1	5	-7	-3	-9	50
496	C3R	P1L	3	5	9	-11	-5	**11**	-1	1	7	-7	-3	-9	50
497	C3R	P1R	3	5	9	-11	-5	**11**	-1	1	7	-7	-3	-9	50
498	C3R	P2C	3	5	9	-11	-3	**11**	-1	1	7	-7	-5	-9	50
499	C3R	P2L	1	7	9	-11	-3	**11**	-1	3	5	-7	-5	-9	50
500	C3R	P2R	3	5	9	-11	-5	**11**	-1	1	7	-7	-3	-9	50
501	C3R	P3C	1	7	9	-11	-5	**11**	-1	3	5	-7	-3	-9	50
502	C3R	P3L	1	5	9	-11	-7	**11**	-1	3	7	-5	-3	-9	50
503	C3R	P3R	3	7	9	-11	-5	**11**	-1	1	5	-7	-3	-9	50
504	C3R	STA	3	7	9	-11	-5	**11**	-1	1	5	-7	-3	-9	50
505	P1C	A1C	5	-10	7	-1	9	**11**	-3	3	-10	-7	-5	1	50
506	P1C	A1L	3	-10	7	-1	9	**11**	-3	5	-10	-7	-5	1	50
507	P1C	A1R	5	-10	9	1	**11**	7	-5	3	-10	-7	-3	-1	50
508	P1C	A2C	3	-10	7	1	9	**11**	-3	5	-10	-7	-5	-1	50
509	P1C	A2L	5	-9	9	-9	7	**11**	-3	3	-9	-5	-1	1	50
510	P1C	A2R	5	-10	9	1	7	**11**	-5	3	-10	-7	-3	-1	50
511	P1C	A3C	5	-10	7	1	**11**	9	-5	3	-10	-7	-3	-1	50
512	P1C	A3L	3	-9	**11**	1	7	9	-3	5	-9	-5	-1	-9	50
513	P1C	A3R	3	-10	9	-1	7	**11**	-3	5	-10	-7	-5	1	50
514	P1C	C1C	5	-10	9	1	7	**11**	-3	3	-10	-7	-5	-1	67
515	P1C	C1L	1	-10	9	-1	7	**11**	-3	5	-10	-5	-7	3	67
516	P1C	C1R	7	-10	9	3	-7	**11**	-1	5	-10	-3	-5	1	67
517	P1C	C2C	5	-10	9	1	7	**11**	-3	3	-10	-7	-5	-1	67
518	P1C	C2L	3	-10	9	1	7	**11**	-3	5	-10	-5	-7	-1	67
519	P1C	C2R	3	-10	7	-1	9	**11**	-3	5	-10	-7	-5	1	67
520	P1C	C3C	5	-9	9	3	-9	**11**	-3	7	-9	-5	1	-1	67
521	P1C	C3L	3	-10	**11**	1	-7	9	-1	7	-10	-5	-3	5	67
522	P1C	C3R	3	-10	9	-1	7	**11**	-3	5	-10	-7	-5	1	67
523	P1C	P1C	5	-10	7	1	9	**11**	-3	3	-10	-7	-5	-1	50
524	P1C	P1L	3	-10	7	1	**11**	9	-5	5	-10	-7	-3	-1	50
525	P1C	P1R	3	-10	7	-1	**11**	9	-5	5	-10	-7	-3	1	50
526	P1C	P2C	5	-10	7	-1	9	**11**	-5	3	-10	-7	-3	1	50
527	P1C	P2L	5	-10	9	-7	7	**11**	-1	3	-10	-5	-3	1	50
528	P1C	P2R	5	-10	7	-1	**11**	9	-5	3	-10	-7	-3	1	50
529	P1C	P3C	3	-10	7	1	**11**	9	-5	5	-10	-7	-3	-1	50
530	P1C	P3L	3	-10	7	1	**11**	9	-5	5	-10	-7	-3	-1	50
531	P1C	P3R	3	-10	7	-1	**11**	9	-5	5	-10	-7	-3	1	50
532	P1C	STA	3	-10	7	1	**11**	9	-1	5	-10	-5	-3	-7	50
533	P1L	A1C	9	-3	3	-1	**11**	7	-7	5	-5	-11	-9	1	50
534	P1L	A1L	9	-3	5	1	**11**	7	-7	3	-5	-11	-9	-1	50
535	P1L	A1R	3	-3	5	1	**11**	9	-7	7	-5	-11	-9	-1	50
536	P1L	A2C	7	-3	5	1	**11**	9	-7	3	-5	-11	-9	-1	50
537	P1L	A2L	3	-3	7	1	**11**	5	-7	9	-5	-11	-9	-1	50

Table A.1: Algorithm wins and losses (continued)

	Objective Space Behaviour	Constraint Space Behaviour	RIGA	HyperM	DDECV	CCSaQPSO	SaQPSO	CCSaDE	SaDE	CCRIGA	CCHyperM	CCGVPSO	GVPSO	CCSaQGVPSO	Problem Changes
538	P1L	A2R	7	-3	5	1	**11**	9	-7	3	-5	-11	-9	-1	50
539	P1L	A3C	7	-3	3	-1	**11**	9	-7	5	-5	-11	-9	1	50
540	P1L	A3L	9	-3	5	1	**11**	7	-7	3	-5	-11	-9	-1	50
541	P1L	A3R	7	-3	5	1	**11**	9	-7	3	-5	-11	-9	-1	50
542	P1L	C1C	9	-3	3	1	**11**	7	-7	5	-5	-9	-11	-1	67
543	P1L	C1L	5	-3	7	3	**11**	9	-7	1	-5	-9	-11	-1	67
544	P1L	C1R	3	-3	7	1	**11**	9	-7	5	-5	-9	-11	-1	67
545	P1L	C2C	7	-3	3	5	**11**	9	-7	1	-5	-9	-11	-1	67
546	P1L	C2L	1	-3	9	3	7	**11**	-7	5	-5	-9	-11	-1	67
547	P1L	C2R	7	-3	5	1	**11**	9	-7	3	-5	-9	-11	-1	67
548	P1L	C3C	7	-3	3	1	**11**	9	-7	5	-5	-9	-11	-1	67
549	P1L	C3L	5	-3	7	3	**11**	9	-7	1	-5	-9	-11	-1	67
550	P1L	C3R	1	-3	5	3	**11**	9	-7	7	-5	-9	-11	-1	67
551	P1L	P1C	9	-3	3	-1	**11**	7	-7	5	-5	-11	-9	1	50
552	P1L	P1L	7	-3	5	-1	**11**	9	-7	3	-5	-11	-9	1	50
553	P1L	P1R	9	-3	5	-1	**11**	7	-7	3	-5	-11	-9	1	50
554	P1L	P2C	9	-3	3	-1	**11**	7	-7	5	-5	-11	-9	1	50
555	P1L	P2L	9	-3	5	-1	**11**	7	-7	3	-5	-11	-9	1	50
556	P1L	P2R	7	-3	3	-1	**11**	9	-7	5	-5	-11	-9	1	50
557	P1L	P3C	9	-3	5	1	**11**	7	-7	3	-5	-11	-9	-1	50
558	P1L	P3L	9	-3	5	1	**11**	7	-7	3	-5	-11	-9	-1	50
559	P1L	P3R	9	-3	5	1	**11**	7	-7	3	-5	-11	-9	-1	50
560	P1L	STA	9	-3	5	1	**11**	7	-7	3	-5	-11	-9	-1	50
561	P1R	A1C	7	-3	9	1	3	**11**	-7	5	-5	-11	-9	-1	50
562	P1R	A1L	7	-5	9	1	5	**11**	-7	3	-3	-11	-9	-1	50
563	P1R	A1R	7	-5	9	1	3	**11**	-7	5	-3	-11	-9	-1	50
564	P1R	A2C	7	-5	9	1	3	**11**	-7	5	-3	-11	-9	-1	50
565	P1R	A2L	5	-5	9	1	3	**11**	-7	7	-3	-11	-9	-1	50
566	P1R	A2R	7	-5	9	1	3	**11**	-7	5	-3	-11	-9	-1	50
567	P1R	A3C	7	-3	9	1	3	**11**	-7	5	-5	-11	-9	-1	50
568	P1R	A3L	7	-5	9	1	3	**11**	-7	5	-3	-11	-9	-1	50
569	P1R	A3R	5	-5	9	1	3	**11**	-7	7	-3	-11	-9	-1	50
570	P1R	C1C	7	-3	9	1	3	**11**	-7	5	-5	-9	-11	-1	67
571	P1R	C1L	7	-5	9	1	3	**11**	-7	5	-3	-9	-11	-1	67
572	P1R	C1R	5	-5	9	1	3	**11**	-7	7	-3	-9	-11	-1	67
573	P1R	C2C	7	-5	9	1	3	**11**	-7	5	-3	-9	-11	-1	67
574	P1R	C2L	5	-5	9	1	3	**11**	-7	7	-3	-9	-11	-1	67
575	P1R	C2R	7	-5	9	1	3	**11**	-7	5	-3	-9	-11	-1	67
576	P1R	C3C	7	-3	9	1	3	**11**	-7	5	-5	-9	-11	-1	67
577	P1R	C3L	7	-5	9	1	3	**11**	-7	5	-3	-9	-11	-1	67
578	P1R	C3R	5	-5	9	1	3	**11**	-7	7	-3	-9	-11	-1	67
579	P1R	P1C	7	-3	9	1	3	**11**	-7	5	-5	-11	-9	-1	50
580	P1R	P1L	7	-5	9	-1	3	**11**	-7	5	-3	-11	-9	1	50
581	P1R	P1R	7	-5	9	-1	3	**11**	-7	5	-3	-11	-9	1	50
582	P1R	P2C	7	-3	9	1	3	**11**	-7	5	-5	-9	-11	-1	50
583	P1R	P2L	7	-3	9	1	3	**11**	-7	5	-5	-11	-9	-1	50
584	P1R	P2R	5	-3	9	1	3	**11**	-7	7	-5	-9	-11	-1	50
585	P1R	P3C	7	-5	9	1	3	**11**	-7	5	-3	-11	-9	-1	50
586	P1R	P3L	7	-5	9	1	3	**11**	-7	5	-3	-11	-9	-1	50
587	P1R	P3R	7	-5	9	1	5	**11**	-7	3	-3	-11	-9	-1	50
588	P1R	STA	7	-5	9	1	5	**11**	-7	3	-3	-11	-9	-1	50
589	P2C	A1C	3	-3	**11**	-1	7	9	-7	5	-5	-11	-9	1	50
590	P2C	A1L	3	-5	9	-1	7	**11**	-7	5	-3	-11	-9	1	50
591	P2C	A1R	1	-5	9	-1	7	**11**	-7	5	-3	-11	-9	3	50
592	P2C	A2C	5	-5	**11**	-1	7	9	-7	3	-3	-11	-9	1	50
593	P2C	A2L	1	-5	9	-1	7	**11**	-7	5	-3	-11	-9	3	50

Table A.1: Algorithm wins and losses (continued)

	Objective Space Behaviour	Constraint Space Behaviour	RIGA	HyperM	DDECV	CCSaQPSO	SaQPSO	CCSaDE	SaDE	CCRIGA	CCHyperM	CCGVPSO	GVPSO	CCSaQGVPSO	Problem Changes
594	P2C	A2R	3	-5	**11**	-1	7	9	-7	5	-3	-11	-9	1	50
595	P2C	A3C	3	-3	9	-1	7	**11**	-7	5	-5	-11	-9	1	50
596	P2C	A3L	3	-5	**11**	-1	7	9	-7	5	-3	-9	-11	1	50
597	P2C	A3R	-1	-3	9	1	7	**11**	-7	5	-5	-11	-9	3	50
598	P2C	C1C	3	-3	**11**	-1	7	9	-7	5	-5	-11	-9	1	67
599	P2C	C1L	1	-5	9	-1	7	**11**	-7	5	-3	-11	-9	3	67
600	P2C	C1R	1	-5	9	-1	7	**11**	-7	5	-3	-11	-9	3	67
601	P2C	C2C	1	-5	9	-1	7	**11**	-7	5	-3	-11	-9	3	67
602	P2C	C2L	1	-5	9	-1	7	**11**	-7	5	-3	-11	-9	3	67
603	P2C	C2R	1	-5	9	-1	7	**11**	-7	5	-3	-11	-9	3	67
604	P2C	C3C	3	-3	9	-1	7	**11**	-7	5	-5	-11	-9	1	67
605	P2C	C3L	1	-5	9	-1	7	**11**	-7	5	-3	-11	-9	3	67
606	P2C	C3R	1	-5	9	-1	7	**11**	-7	5	-3	-11	-9	3	67
607	P2C	P1C	3	-3	**11**	-1	7	9	-7	5	-5	-11	-9	1	50
608	P2C	P1L	3	-5	9	1	7	**11**	-7	5	-3	-11	-9	-1	50
609	P2C	P1R	3	-5	9	1	7	**11**	-7	5	-3	-11	-9	-1	50
610	P2C	P2C	1	-3	9	-1	7	**11**	-7	5	-5	-11	-9	3	50
611	P2C	P2L	1	-5	9	-1	7	**11**	-7	5	-3	-11	-9	3	50
612	P2C	P2R	3	-3	9	-1	7	**11**	-7	5	-5	-11	-9	1	50
613	P2C	P3C	3	-5	9	-1	7	**11**	-7	5	-3	-11	-9	1	50
614	P2C	P3L	3	-5	9	1	7	**11**	-7	5	-3	-11	-9	-1	50
615	P2C	P3R	3	-3	9	1	7	**11**	-7	5	-5	-11	-9	-1	50
616	P2C	STA	-1	-3	9	1	7	**11**	-7	5	-5	-11	-9	3	50
617	P2L	A1C	1	-5	3	-1	9	5	-7	7	-3	-9	-11	**11**	50
618	P2L	A1L	1	-5	7	-1	**11**	5	-7	3	-3	-9	-11	9	50
619	P2L	A1R	1	-5	7	-1	9	5	-7	3	-3	-9	-11	**11**	50
620	P2L	A2C	1	-5	3	-1	9	5	-7	7	-3	-9	-11	**11**	50
621	P2L	A2L	-1	-5	3	1	9	5	-7	7	-3	-9	-11	**11**	50
622	P2L	A2R	1	-5	3	-1	9	5	-7	7	-3	-9	-11	**11**	50
623	P2L	A3C	1	-5	5	-1	9	3	-7	7	-3	-9	-11	**11**	50
624	P2L	A3L	-1	-5	3	1	**11**	5	-7	7	-3	-9	-11	9	50
625	P2L	A3R	1	-5	3	-1	9	5	-7	7	-3	-9	-11	**11**	50
626	P2L	C1C	1	-5	3	-1	9	5	-7	7	-3	-11	-9	**11**	67
627	P2L	C1L	1	-5	5	-1	9	3	-7	7	-3	-11	-9	**11**	67
628	P2L	C1R	-1	-5	5	1	9	3	-7	7	-3	-11	-9	**11**	67
629	P2L	C2C	-1	-5	3	1	9	7	-7	5	-3	-11	-9	**11**	67
630	P2L	C2L	-1	-5	3	1	7	5	-7	9	-3	-9	-11	**11**	67
631	P2L	C2R	-1	-5	3	1	9	7	-7	5	-3	-11	-9	**11**	67
632	P2L	C3C	1	-5	5	-1	9	3	-7	7	-3	-11	-9	**11**	67
633	P2L	C3L	-1	-5	5	1	9	3	-7	7	-3	-11	-9	**11**	67
634	P2L	C3R	-1	-5	3	1	7	5	-7	9	-3	-11	-9	**11**	67
635	P2L	P1C	1	-5	3	-1	9	5	-7	7	-3	-9	-11	**11**	50
636	P2L	P1L	3	-5	7	-1	**11**	9	-7	1	-3	-9	-11	5	50
637	P2L	P1R	1	-5	5	-1	**11**	9	-7	3	-3	-9	-11	7	50
638	P2L	P2C	1	-3	3	-1	**11**	7	-7	5	-5	-9	-11	9	50
639	P2L	P2L	1	-5	3	-1	9	7	-7	5	-3	-9	-11	**11**	50
640	P2L	P2R	-1	-3	3	1	**11**	7	-7	5	-5	-9	-11	9	50
641	P2L	P3C	1	-5	5	-1	**11**	9	-7	3	-3	-9	-11	7	50
642	P2L	P3L	3	-5	5	-1	**11**	9	-7	1	-3	-9	-11	7	50
643	P2L	P3R	3	-5	5	-1	**11**	9	-7	1	-3	-9	-11	7	50
644	P2L	STA	1	-5	3	-1	**11**	5	-7	9	-3	-9	-11	7	50
645	P2R	A1C	1	-5	5	-1	**11**	7	-7	3	-3	-11	-9	9	50
646	P2R	A1L	1	-5	9	-1	**11**	5	-7	3	-3	-11	-9	7	50
647	P2R	A1R	1	-5	7	-1	9	5	-7	3	-3	-11	-9	**11**	50
648	P2R	A2C	1	-5	7	-1	**11**	5	-7	3	-3	-11	-9	9	50
649	P2R	A2L	-1	-5	7	1	9	5	-7	3	-3	-11	-9	**11**	50

Table A.1: Algorithm wins and losses (continued)

	Objective Space Behaviour	Constraint Space Behaviour	RIGA	HyperM	DDECV	CCSaQPSO	SaQPSO	CCSaDE	SaDE	CCRIGA	CCHyperM	CCGVPSO	GVPSO	CCSaQGVPSO	Problem Changes
650	P2R	A2R	1	-5	7	-1	**11**	3	-7	5	-3	-11	-9	9	50
651	P2R	A3C	1	-5	7	-1	**11**	5	-7	3	-3	-11	-9	9	50
652	P2R	A3L	-1	-5	5	1	**11**	9	-7	3	-3	-11	-9	7	50
653	P2R	A3R	1	-5	5	-1	**11**	7	-7	3	-3	-11	-9	9	50
654	P2R	C1C	1	-5	5	-1	**11**	7	-7	3	-3	-11	-9	9	67
655	P2R	C1L	1	-5	5	-1	9	7	-7	3	-3	-11	-9	**11**	67
656	P2R	C1R	1	-5	7	-1	9	5	-7	3	-3	-11	-9	**11**	67
657	P2R	C2C	-1	-5	7	1	9	5	-7	3	-3	-11	-9	**11**	67
658	P2R	C2L	1	-5	9	-1	3	7	-7	5	-3	-11	-9	**11**	67
659	P2R	C2R	-1	-5	7	1	9	3	-7	5	-3	-11	-9	**11**	67
660	P2R	C3C	1	-5	7	-1	**11**	5	-7	3	-3	-11	-9	9	67
661	P2R	C3L	-1	-5	7	1	3	9	-7	5	-3	-11	-9	**11**	67
662	P2R	C3R	-1	-5	3	1	9	7	-7	5	-3	-11	-9	**11**	67
663	P2R	P1C	1	-5	5	-1	**11**	7	-7	3	-3	-11	-9	9	50
664	P2R	P1L	3	-5	7	-1	**11**	9	-7	1	-3	-9	-11	5	50
665	P2R	P1R	3	-5	7	-1	**11**	9	-7	1	-3	-9	-11	5	50
666	P2R	P2C	1	-3	7	-1	**11**	5	-7	3	-5	-11	-9	9	50
667	P2R	P2L	1	-5	7	-1	**11**	5	-7	3	-3	-11	-9	9	50
668	P2R	P2R	1	-3	9	-1	**11**	5	-7	3	-5	-11	-9	7	50
669	P2R	P3C	3	-3	7	-1	**11**	9	-7	1	-5	-9	-11	5	50
670	P2R	P3L	3	-3	7	-1	**11**	9	-7	1	-5	-9	-11	5	50
671	P2R	P3R	3	-3	7	-1	**11**	9	-7	1	-5	-9	-11	5	50
672	P2R	STA	3	-3	7	-1	**11**	9	-7	1	-5	-9	-11	5	50
673	P3C	A1C	5	-3	7	1	**11**	9	-7	3	-5	-9	-11	-1	50
674	P3C	A1L	5	-5	7	1	**11**	9	-7	3	-3	-11	-9	-1	50
675	P3C	A1R	5	-5	7	3	**11**	9	-7	1	-3	-9	-11	-1	50
676	P3C	A2C	5	-3	9	1	7	**11**	-7	3	-5	-9	-11	-1	50
677	P3C	A2L	5	-3	9	3	7	**11**	-7	1	-5	-9	-11	-1	50
678	P3C	A2R	5	-3	9	1	7	**11**	-7	3	-5	-9	-11	-1	50
679	P3C	A3C	5	-3	7	1	**11**	9	-7	3	-5	-9	-11	-1	50
680	P3C	A3L	5	-5	7	1	**11**	9	-7	3	-3	-9	-11	-1	50
681	P3C	A3R	5	-5	7	3	9	**11**	-7	1	-3	-9	-11	-1	50
682	P3C	C1C	5	-3	7	1	**11**	9	-7	3	-5	-9	-11	-1	67
683	P3C	C1L	5	-5	7	3	9	**11**	-7	1	-3	-9	-11	-1	67
684	P3C	C1R	5	-5	9	1	7	**11**	-7	3	-3	-9	-11	-1	67
685	P3C	C2C	3	-3	7	5	9	**11**	-7	1	-5	-9	-11	-1	67
686	P3C	C2L	3	-3	9	1	7	**11**	-7	5	-5	-9	-11	-1	67
687	P3C	C2R	5	-3	7	1	9	**11**	-7	3	-5	-9	-11	-1	67
688	P3C	C3C	5	-3	7	1	**11**	9	-7	3	-5	-9	-11	-1	67
689	P3C	C3L	5	-5	9	1	7	**11**	-7	3	-3	-9	-11	-1	67
690	P3C	C3R	5	-5	9	3	7	**11**	-7	1	-3	-9	-11	-1	67
691	P3C	P1C	5	-3	7	1	**11**	9	-7	3	-5	-9	-11	-1	50
692	P3C	P1L	5	-5	7	1	**11**	9	-7	3	-3	-9	-11	-1	50
693	P3C	P1R	5	-5	7	1	**11**	9	-7	3	-3	-11	-9	-1	50
694	P3C	P2C	5	-3	9	1	7	**11**	-7	3	-5	-9	-11	-1	50
695	P3C	P2L	5	-3	7	1	9	**11**	-7	3	-5	-9	-11	-1	50
696	P3C	P2R	5	-5	9	1	7	**11**	-7	3	-3	-9	-11	-1	50
697	P3C	P3C	5	-5	7	1	**11**	9	-7	3	-3	-9	-11	-1	50
698	P3C	P3L	5	-5	7	1	**11**	9	-7	3	-3	-9	-11	-1	50
699	P3C	P3R	5	-5	7	1	**11**	9	-7	3	-3	-9	-11	-1	50
700	P3C	STA	5	-5	9	1	**11**	7	-7	3	-3	-9	-11	-1	50
701	P3L	A1C	3	-3	7	1	**11**	9	-7	5	-5	-9	-11	-1	50
702	P3L	A1L	5	-3	7	1	**11**	9	-7	3	-5	-9	-11	-1	50
703	P3L	A1R	3	-3	7	1	**11**	9	-7	5	-5	-9	-11	-1	50
704	P3L	A2C	5	-3	9	1	7	**11**	-7	3	-5	-9	-11	-1	50
705	P3L	A2L	1	-3	9	3	7	**11**	-7	5	-5	-9	-11	-1	50

Table A.1: Algorithm wins and losses (continued)

	Objective Space Behaviour	Constraint Space Behaviour	RIGA	HyperM	DDECV	CCSaQPSO	SaQPSO	CCSaDE	SaDE	CCRIGA	CCHyperM	CCGVPSO	GVPSO	CCSaQGVPSO	Problem Changes
706	P3L	A2R	5	-3	9	1	7	**11**	-7	3	-5	-9	-11	-1	50
707	P3L	A3C	3	-3	7	1	**11**	9	-7	5	-5	-9	-11	-1	50
708	P3L	A3L	3	-3	7	1	**11**	9	-7	5	-5	-9	-11	-1	50
709	P3L	A3R	3	-3	7	1	9	**11**	-7	5	-5	-9	-11	-1	50
710	P3L	C1C	3	-3	7	1	**11**	9	-7	5	-5	-9	-11	-1	67
711	P3L	C1L	5	-3	7	1	**11**	9	-7	3	-5	-9	-11	-1	67
712	P3L	C1R	3	-3	9	1	7	**11**	-7	5	-5	-9	-11	-1	67
713	P3L	C2C	5	-3	7	1	9	**11**	-7	3	-5	-9	-11	-1	67
714	P3L	C2L	1	-3	9	3	7	**11**	-7	5	-5	-9	-11	-1	67
715	P3L	C2R	1	-3	7	3	9	**11**	-7	5	-5	-9	-11	-1	67
716	P3L	C3C	3	-3	7	1	**11**	9	-7	5	-5	-9	-11	-1	67
717	P3L	C3L	3	-3	9	1	7	**11**	-7	5	-5	-9	-11	-1	67
718	P3L	C3R	3	-3	9	1	7	**11**	-7	5	-5	-9	-11	-1	67
719	P3L	P1C	3	-3	7	1	**11**	9	-7	5	-5	-9	-11	-1	50
720	P3L	P1L	5	-3	7	1	**11**	9	-7	3	-5	-11	-9	-1	50
721	P3L	P1R	5	-3	7	1	**11**	9	-7	3	-5	-11	-9	-1	50
722	P3L	P2C	3	-3	9	1	7	**11**	-7	5	-5	-9	-11	-1	50
723	P3L	P2L	3	-3	7	1	9	**11**	-7	5	-5	-9	-11	-1	50
724	P3L	P2R	5	-3	7	1	9	**11**	-7	3	-5	-9	-11	-1	50
725	P3L	P3C	5	-3	7	1	**11**	9	-7	3	-5	-9	-11	-1	50
726	P3L	P3L	5	-3	7	1	**11**	9	-7	3	-5	-9	-11	-1	50
727	P3L	P3R	3	-3	7	1	**11**	9	-7	5	-5	-11	-9	-1	50
728	P3L	STA	5	-3	7	1	**11**	9	-7	3	-5	-11	-9	-1	50
729	P3R	A1C	7	-5	9	1	5	**11**	-7	3	-3	-11	-9	-1	50
730	P3R	A1L	5	-5	9	1	7	**11**	-7	3	-3	-11	-9	-1	50
731	P3R	A1R	5	-5	9	1	7	**11**	-7	3	-3	-9	-11	-1	50
732	P3R	A2C	3	-5	9	1	5	**11**	-7	7	-3	-11	-9	-1	50
733	P3R	A2L	5	-3	9	1	3	**11**	-7	7	-5	-11	-9	-1	50
734	P3R	A2R	5	-5	9	1	3	**11**	-7	7	-3	-9	-11	-1	50
735	P3R	A3C	5	-5	9	1	7	**11**	-7	3	-3	-11	-9	-1	50
736	P3R	A3L	5	-5	9	1	7	**11**	-7	3	-3	-11	-9	-1	50
737	P3R	A3R	5	-5	9	1	7	**11**	-7	3	-3	-9	-11	-1	50
738	P3R	C1C	5	-5	9	1	7	**11**	-7	3	-3	-11	-9	-1	67
739	P3R	C1L	5	-5	9	1	7	**11**	-7	3	-3	-11	-9	-1	67
740	P3R	C1R	7	-5	9	1	5	**11**	-7	3	-3	-9	-11	-1	67
741	P3R	C2C	3	-5	9	1	7	**11**	-7	5	-3	-11	-9	-1	67
742	P3R	C2L	5	-3	9	1	3	**11**	-7	7	-5	-11	-9	-1	67
743	P3R	C2R	3	-5	9	1	5	**11**	-7	7	-3	-11	-9	-1	67
744	P3R	C3C	5	-5	9	1	7	**11**	-7	3	-3	-11	-9	-1	67
745	P3R	C3L	5	-5	9	1	7	**11**	-7	3	-3	-11	-9	-1	67
746	P3R	C3R	7	-5	9	1	5	**11**	-7	3	-3	-11	-9	-1	67
747	P3R	P1C	7	-5	9	1	5	**11**	-7	3	-3	-11	-9	-1	50
748	P3R	P1L	7	-5	9	1	5	**11**	-7	3	-3	-11	-9	-1	50
749	P3R	P1R	7	-5	9	1	5	**11**	-7	3	-3	-11	-9	-1	50
750	P3R	P2C	3	-3	9	1	5	**11**	-7	7	-5	-9	-11	-1	50
751	P3R	P2L	7	-3	9	1	3	**11**	-7	5	-5	-11	-9	-1	50
752	P3R	P2R	5	-5	9	1	3	**11**	-7	7	-3	-11	-9	-1	50
753	P3R	P3C	7	-5	9	1	5	**11**	-7	3	-3	-11	-9	-1	50
754	P3R	P3L	5	-5	9	1	7	**11**	-7	3	-3	-11	-9	-1	50
755	P3R	P3R	5	-5	9	1	7	**11**	-7	3	-3	-11	-9	-1	50
756	P3R	STA	5	-5	9	1	7	**11**	-7	3	-3	-11	-9	-1	50
757	STA	A1C	7	-5	5	-1	**11**	9	-3	3	-9	-7	-11	1	10
758	STA	A1L	9	-5	5	-1	**11**	7	-3	3	-9	-7	-11	1	10
759	STA	A1R	7	-5	5	-1	**11**	9	-3	3	-9	-7	-11	1	10
760	STA	A2C	7	-9	5	-1	**11**	9	-3	3	-5	-7	-11	1	10
761	STA	A2L	3	-7	7	-1	**11**	9	-3	5	-5	-9	-11	1	10

Table A.1: Algorithm wins and losses (continued)

	Objective Space Behaviour	Constraint Space Behaviour	RIGA	HyperM	DDECV	CCSaQPSO	SaQPSO	CCSaDE	SaDE	CCRIGA	CCHyperM	CCGVPSO	GVPSO	CCSaQGVPSO	Problem Changes
762	STA	A2R	7	-5	5	-1	9	**11**	-3	3	-7	-9	-11	1	10
763	STA	A3C	7	-5	5	-1	**11**	9	-3	3	-9	-7	-11	1	10
764	STA	A3L	9	-5	5	-1	**11**	7	-3	3	-9	-7	-11	1	10
765	STA	A3R	7	-5	5	-1	**11**	9	-3	3	-7	-9	-11	1	10
766	STA	C1C	7	-3	5	-1	**11**	9	-7	3	-5	-9	-11	1	33
767	STA	C1L	9	-3	5	1	**11**	7	-7	3	-5	-9	-11	-1	33
768	STA	C1R	7	-3	3	1	9	**11**	-7	5	-5	-9	-11	-1	33
769	STA	C2C	5	-5	7	-1	**11**	9	-7	3	-3	-9	-11	1	33
770	STA	C2L	3	-5	9	-1	7	**11**	-7	5	-3	-9	-11	1	33
771	STA	C2R	3	-5	7	-1	9	**11**	-7	5	-3	-9	-11	1	33
772	STA	C3C	7	-3	5	-1	**11**	9	-7	3	-5	-9	-11	1	33
773	STA	C3L	5	-3	7	-1	**11**	9	-7	3	-5	-9	-11	1	33
774	STA	C3R	7	-3	5	-1	**11**	9	-7	3	-5	-9	-11	1	33
775	STA	P1C	7	-3	5	1	**11**	9	-7	3	-5	-9	-11	-1	50
776	STA	P1L	9	-3	5	-1	**11**	7	-7	3	-5	-9	-11	1	50
777	STA	P1R	9	-5	5	-1	**11**	7	-7	3	-3	-9	-11	1	50
778	STA	P2C	3	-3	7	-1	**11**	9	-7	5	-5	-9	-11	1	50
779	STA	P2L	7	-5	5	-1	**11**	9	-7	3	-3	-9	-11	1	50
780	STA	P2R	3	-3	7	-1	**11**	9	-7	5	-5	-9	-11	1	50
781	STA	P3C	7	-5	5	-1	**11**	9	-7	3	-3	-9	-11	1	50
782	STA	P3L	7	-5	5	-1	**11**	9	-7	3	-3	-9	-11	1	50
783	STA	P3R	7	-5	5	-1	**11**	9	-7	3	-3	-9	-11	1	50
784	STA	STA	5	-11	3	-7	**11**	7	9	1	-9	-1	-3	-5	1

Appendix B

Acronyms

ABC artificial bee colony 47

ABEAC average best error after change 96, 97, 122, 123, 145, 184

ABEBC average best error before change 95–97, 122–124, 145, 184

ADT algebraic data type 234

API application programming interface 208, 212, 216

ARR absolute recovery rate 96, 97, 145, 184

BBPSO bare-bones particle swarm optimisation 38

BOG best of generation 90, 91, 93, 96

CCGVPSO cooperative co-evolutionary gaussian-valued particle swarm optimisation 134, 141, 142, 149, 150, 153, 157, 160, 175, 182, 185

CCHyperM cooperative co-evolutionary hyper mutation genetic algorithm 133, 141, 142, 149, 150, 155, 162, 163, 171, 175, 182, 185–187, 193

CCPSO cooperative co-evolutionary particle swarm optimisation x, 128–131, 134, 137

CCRIGA cooperative co-evolutionary random immigrant genetic algorithm 133, 141, 142, 149, 150, 155, 160, 190, 192, 193

CCSaDE cooperative co-evolutionary self-adaptive differential evolution 133, 134, 141, 142, 149, 150, 153, 155, 157, 160–164, 170–172, 175, 182, 185, 190, 192, 193

CCSaQGVPSO cooperative co-evolutionary self-adaptive quantum gaussian-valued particle swarm optimisation 134, 141, 142, 155, 157, 160, 162, 182

CCSaQPSO cooperative co-evolutionary self-adaptive quantum paritcle swarm optimisation 134, 141, 142, 149, 150, 153, 155, 160, 162, 164

CDF cumulative distribution function 230

CE cultural evolution 46, 56

CEC Congress on Evolutionary Computation 70

CI computational intelligence 1, 5, 90, 101, 146, 196, 197, 201, 212, 214, 239, 241, 244, 245

CIlib computational intelligence library 204

CLI command line interface 208

CME collective mean error 91, 122–124

CMF collective mean fitness 91

CMPB constrained moving peaks benchmark xi, 3, 83, 84, 86, 88, 112, 132, 139, 140, 145, 163, 178, 244

CMWC complementary multiply with carry 229

CoEA co-evolutionary algorithm 47, 48

COP constrained optimisation problem 10, 21, 25

CPSO charged particle swarm optimisation 43, 47, 51

CPU central processing unit 206

CSV comma separated values 199

DCBG dynamic composition benchmark generator 68, 69

DCOP dynamic constrained optimisation problem xi, 2–5, 13, 14, 27, 28, 59, 60, 70–72, 79–81, 88, 102, 111, 113, 115, 127–133, 135–141, 144, 145, 148, 149, 155–161, 164, 171, 179, 185, 190, 192–194, 242–245

DDECV dynamic differential evolution with combined variants 50, 51, 141, 142, 149, 150, 153, 155, 157, 160–164, 170–172, 175, 182, 185, 190, 193

DE differential evolution 2, 30, 33–35, 39, 46, 48, 50, 51, 134, 186

DODC dynamic objective function with dynamic constraints 14, 27, 70, 88, 131, 139, 140, 155–157, 161, 192, 193, 242, 243

DOP dynamic optimisation problem 2, 4, 10, 11, 13–16, 27, 50, 54, 55, 60, 61, 72, 79–81, 88–90, 92, 94, 97, 98, 102, 103, 107–109, 111–113, 115, 117, 118, 133, 134, 136, 141, 142, 145, 193, 243, 244

DOSC dynamic objective function with static constraints 14, 27, 88, 131, 139, 140, 155–157, 192, 242

DynDE dynamic differential evolution 46, 47

EA evolutionary algorithm 33, 210

EC evolutionary computation 201, 204, 206, 208, 209, 212, 215, 219, 235, 242

EP evolutionary programming 46

ES evolutionary strategy 26, 46

FCI fitness cloud index 77, 86, 87

FDC fitness distance correlation 78, 86, 87

FEM first entropic measure 77, 86, 87

FFI foreign function interface 204

FLA fitness landscape analysis 72, 80, 88, 105, 243

FP functional programming 218–220

FSR feasibility ratio 76, 86

GA genetic algorithm 2, 30, 31, 33, 39, 42, 47, 48, 133, 186, 228, 235

GCPSO guaranteed convergence particle swarm optimizer 233

GDBG generalized dynamic benchmark generator 66, 67

GP genetic programming 41, 219

GPU graphics processing unit 206, 211

GUI graphical user interface 207, 208

GVPSO gaussian-valued particle swarm optimisation 38, 39, 134, 141, 142, 149, 150, 153, 155, 157, 160, 175, 182

HBEBC highest best error before change 96

HKT higher-kinded type 225

HyperM hyper-mutation genetic algorithm 48, 50, 133, 141, 142, 149, 150, 162, 163, 165, 168, 170, 171, 175, 182, 185–187, 193

JSON JavaScript Object Notation 199, 200

LBEBC lowest best error before change 96

LCG linear congruential generator 202

LEAP Library for Evolutionary Algorithms in Python 204, 206

MOE modified off-line error 92, 93

MOO multi-objective optimisation 20

MOP modified off-line performance 92

MPB moving peaks benchmark 18, 62–65, 67, 69, 79, 80, 82, 84, 86, 88, 112, 121, 140, 178

NFL no free lunch theorem 89

OOP object-oriented programming 218, 220, 239

OP on-line performance 91, 92

PCX parent centric crossover 32, 116

PRNG pseudo-random number generator 202, 204, 206, 208, 210, 211, 217, 229, 230, 232, 239

PSO particle swarm optimisation 2, 4, 26, 30, 35, 36, 38, 39, 41–45, 47, 51, 53, 55, 56, 77, 115, 121, 129–131, 134, 143, 186, 228, 232, 235, 238, 239

QPSO quantum particle swarm optimisation 5, 51, 53, 115–117, 120–124, 127, 186

RDBG rotation dynamic benchmark generator 67, 69

REPL read-evaluate-proint loop 230

RFBx ratio feasibility boundary crossings 76, 86, 87

RIGA random immigrants genetic algorithm 48, 50, 133, 138, 141, 142, 149, 150, 155, 160, 161, 190, 192, 193

SaDE self-adaptive differential evolution 35, 133, 134, 141, 142, 150, 153, 155, 157, 162, 163, 165, 168, 170, 175, 182, 185, 186, 192, 193

SaQGVPSO self-adaptive quantum gaussian-valued particle swarm optimisation 134

SaQPSO self-adaptive quantum particle swarm optimisation 4, 119–127, 134, 137, 141, 142, 149, 150, 153, 155, 157, 160–162, 164–166, 168, 170, 172, 182, 186, 192, 193

SI swarm intelligence 201, 204, 208, 209, 212, 215, 219, 235, 242

SODC static objective function with dynamic constraints 14, 27, 88, 131, 139, 140, 155, 156, 192, 242

SOP static optimisation problem 2, 9, 35, 39, 54, 89, 109, 134, 243, 244

SOSC static objective function with static constraints 13, 24, 26, 27, 88, 131, 132, 137, 139, 140, 155, 156, 192, 242

WEKA Waikato Environment for Knowledge Analysis 204, 207, 208

Appendix C

Derived Publications

This section provides a list of all published and submitted conference and journal articles derived from the content of this thesis.

The following is a list of published work:

- G. Pamparà and A. P. Engelbrecht. "Towards A Generic Computational Intelligence Library: Preventing Insanity". In: *Proceedings of the IEEE Symposium Series on Computational Intelligence*. 2015 IEEE Symposium Series on Computational Intelligence. Dec. 2015, pp. 1460–1467. DOI: 10.1109/SSCI.2015.207
- G. Pamparà and A. P. Engelbrecht. "Self-adaptive Quantum Particle Swarm Optimization for Dynamic Environments". In: *Proceedings of the International Conference on Swarm Intelligence*. Ed. by M. Dorigo *et al.* 2018, pp. 163–175. DOI: 10.1007/978-3-030-00533-7_13
- G. Pamparà and A. P. Engelbrecht. "Evolutionary and Swarm Intelligence Algorithms Through Monadic Composition". In: *Proceedings of the Genetic and Evolutionary Computation Conference Companion*. GECCO '19. July 13, 2019, pp. 1382–1390. DOI: 10.1145/3319619.3326845
- G. Pamparà and A. P. Engelbrecht. "A Generator for Dynamically Constrained Optimization Problems". In: *Proceedings of the Genetic and Evolutionary Computation Conference Companion*. GECCO '19. July 13, 2019, pp. 1441–1448. DOI: 10.1145/3319619.3326798

The following work has been submitted for review:

- Performance Analysis of Dynamic Optimisation Algorithms Using Relative Error Distance, Swarm and Evolutionary Computation, 2021

The following is planned for submission:

- Computational Intelligence with Certainty Through Functional Programming, Journal of Systems and Software, 2021