Population-Based Runtime Optimisation in Static and Dynamic Environments

Von der Fakultät für Elektrotechnik und Informatik
der Gottfried Wilhelm Leibniz Universität Hannover
zur Erlangung des akademischen Grades
Doktor-Ingenieur
genehmigte Dissertation

von M. Sc. Emre Cakar

geboren am 24. Mai 1979 in Gaziantep

2011

1. Referent: Prof. Dr.-Ing. Christian Müller-Schloer
2. Referent: Prof. Dr.-Ing. Bernardo Wagner
Tag der Promotion: 13.09.2011

Zusammenfassung

Schlagworte: Organic Computing, populationsbasierte Optimierung, statische and dynamische Fitnesslandschaften

Adaptivität und Robustheit sind die wesentlichen Merkmale moderner technischer Systeme, die aus einer Vielzahl von miteinander kooperierenden Subsystemen bestehen und in der Lage sind, bestimmte Aufgaben gemeinsam zu bewältigen. Unterschiedliche Randbedingungen, wie z.B. die Veränderungen in der Umwelt oder bestimmte Störungen, können zur Folge haben, dass das System auf solche Effekte nicht adäquat reagiert und somit nicht auf dem gewünschten Leistungsniveau arbeitet. Diese Tatsache erfordert geeignete Mechanismen, die es einem technischen System ermöglichen, sein Verhalten (und gegebenenfalls auch seine Struktur) an solche veränderlichen Umgebungen anzupassen. Das Ziel des Organic Computing (OC) ist, technische Systeme mit lebensähnlichen Eigenschaften, wie z.B. der Selbstorganisation, Selbstkonfiguration und Selbstadaptation, auszustatten, um die Funktionsfähigkeit dieser Systeme zu garantieren und zu jedem Zeitpunkt die optimale Systemleistung zu erzielen. In diesem Zusammenhang spielt die Selbstadaptation eine ganz wesentliche Rolle. Diese erfordert die Nutzung geeigneter Optimierungsverfahren, die in der Lage sind, in kürzester Zeit hoch qualitative Lösungen zu liefern, um in statischen, verrauschten oder dynamischen Umgebungen eine schnelle Anpassung des Systemverhaltens und somit eine optimale Leistung ermöglichen.

In dieser Arbeit wird ein neues populationsbasiertes Optimierungsverfahren (Role-based Imitation Algorithm - RBI) vorgestellt, das die Ermittlung der (möglichst) optimalen Lösung für OC Systeme mit statischen und dynamischen Fitnesslandschaften gewährleistet. RBI benutzt ein neuartiges Rollenzuweisungsverfahren für explorierende und exploitierende Individuen einer Population, um eine intelligente und schnelle Suche nach dem Optimum in der gegebenen Fitnesslandschaft zu realisieren. In dem vorgestellten Verfahren findet für jeden Optimierungsschritt eine klare Trennung der explorierenden und exploitierenden Individuen statt. Diese sorgt dafür, dass die Gesamt-

population gleichzeitig nach besseren Lösungen in der Fitnesslandschaft sucht, ohne dabei die bereits gefundenen guten Lösungen zu vergessen. In dieser Arbeit wird RBI mit bekannten Verfahren aus der Literatur (wie z.B. Differential Evolution, Genetic Algorithms und Particle Swarm Optimisation) in unterschiedlichen statischen (mit und ohne Rauschen) und dynamischen Fitnesslandschaften verglichen. Zur Untersuchung in statischen Fitnesslandschaften werden diverse unimodale und multimodale Benchmark-Funktionen mit unterschiedlicher Anzahl von Dimensionen aus dem Themengebiet Funktionsoptimierung verwendet. Um die Leistung des RBI in dynamischen Fitnesslandschaften zu testen, werden bestimmte Anwendungsszenarien aus der Jäger-Beute-Domäne verwendet. Die Jäger-Beute-Domäne stellt ein generisches Modell für eine Vielzahl von Multiagentensystemen zur Verfügung, wo die Agenten miteinander kooperieren um bestimmte Aufgaben zu erledigen, was wiederum komplexe dynamische Optimierungsprobleme zur Folge hat.

Für den Vergleich von RBI mit anderen Optimierungsverfahren werden unterschiedliche Experimente durchgeführt, um (1) die Qualität der gefundenen Lösungen und (2) die Konvergenzgeschwindigkeit der Verfahren in der jeweiligen Fitnesslandschaft zu testen. Die Experimente zeigen, dass RBI besonders in statischen verrauschten und dynamischen Fitnesslandschaften bessere Ergebnisse als die anderen Optimierungsverfahren erzielt und somit ein hohes Maßan Adaptivität und Robustheit gewährleistet.

Basierend auf den Ergebnissen werden unterschiedliche Möglichkeiten zur Parallelisierung des RBI-Verfahrens diskutiert.

Abstract

Keywords: Organic Computing, population-based optimisation, static and dynamic fitness landscapes

Adaptivity and robustness are key concepts in developing today‘s technical systems. An increasing number of system elements, their complexity and a dynamically changing environment often lead to unexpected system behaviour, although all system elements are available and work correctly. This requires adequate mechanisms in order to provide a technical system with capabilities to adapt its behaviour to new environmental situations and work properly towards its predefined goal. The vision of Organic Computing (OC) is to endow technical systems with life-like properties such as self-organisation, self-configuration and self-adaptation in order to address this complexity. In this context, self-adaptation is a key aspect that allows a system to perform well in a (possibly dynamic) environment without intervention from outside. Establishing self-adaptation in technical systems requires adequate optimisation algorithms that can find high-quality solutions in a short time.

This thesis presents a new population-based optimisation algorithm, the Role-based Imitation Algorithm (RBI), that can be used to establish self-adaptation in OC systems with static and dynamic fitness landscapes. RBI proposes a novel role assignment strategy in order to provide a strict distinction between the exploring and exploiting individuals of the population. This role assignment takes place according to the convergence of solutions represented by the individuals and to the corresponding fitness values, which facilitates an effective optimisation scheme, where previously found good solutions are kept while other parts of the fitness landscape are further explored simultaneously. In this thesis, we investigate different problem settings with static (noiseless/noisy) and dynamic fitness landscapes and evaluate the performance of RBI in comparison to state-of-the-art optimisation algorithms from the literature such as Differential Evolution, Genetic Algorithms and Particle Swarm Optimisation. For the comparison in static fitness landscapes, we use noiseless and noisy benchmark functions from the literature each with different proper-

ties regarding the essential aspects of fitness functions such as multimodality, high dimensionality and separability. In order to investigate the performance of RBI in dynamic fitness landscapes, we consider different scenarios from the pursuit (predator/prey) domain, since these scenarios represent a generic model for many multi-agent systems (MAS), which consist of agents moving around in an environment and interacting with each other in order to accomplish a given task resulting in dynamic optimisation problems. Different experiments are carried out in order to determine (1) the solution quality and (2) the convergence speed obtained by RBI in comparison to other optimisation algorithms. Our experiments show that RBI outperforms its competitors especially in noisy and highly dynamic environments providing a high level of robustness and adaptivity.

Finally, based on the presented experimental results the future research opportunities towards the parallelisation of RBI are discussed.

Contents

List of Figures

List of Algorithms

List of Abbreviations

OC	Organic Computing
SuOC	System under Observation and Control
XCS	Extended Classifier System
OTC	Organic Traffic Control
BF	Bit-flip
CX	Complementary Crossover
GPS	Generalised Pattern Search
SA	Simulated Annealing
HC	Hill Climbing
EA	Evolutionary Algorithm
GA	Genetic Algorithm
DE	Differential Evolution
LCS	Learning Classifier System
GP	Genetic Programming
EP	Evolutionary Computing
PSO	Particle Swarm Optimisation
ACO	Ant Colony Optimisation
RBI	Role-based Imitation algorithm
SI	Swarm Intelligence
CSP	Constraint Satisfaction Problem
SAT	Boolean Satisfiability Problem
SPX	Single-point Crossover
MPX	Multi-point Crossover
TSP	Traveling Salesman Problem
PMX	Partially Match Crossover

MAS	Multi-agent Systems
TS	Target Space
AS	Acceptance Space
SS	Survival Space
DS	Dead Space
LS	Local Search

List of Publications

1. **Towards a quantitative notion of self-organisation**
   Emre Cakar, Moez Mnif, Christian Müller-Schloer, Urban Richter, and Hartmut Schmeck
   In IEEE Congress on Evolutionary Computation, 2007. CEC 2007. September 2007, pp. 4222-4229.

2. **Creating Collaboration Patterns in Multi-Agent Systems with Generic Observer/Controller Architectures**
   Emre Cakar, Jörg Hähner, and Christian Müller-Schloer
   In Proceedings of the 2nd International Conference on Autonomic Computing and Communication Systems, ICST, Brussels, Belgium, 2008, Autonomics '08, pp. 6:1-6:9

3. **Investigation of Generic Observer/Controller Architectures in a Traffic Scenario**
   Emre Cakar, Jörg Hähner, and Christian Müller-Schloer
   In GI Jahrestagung (2), 2008, Vol. 134GI , pp. 733-738.

4. **Dynamic Control of Network Protocols - a new vision for future self-orgsanised networks**
   Sven Tomforde, Emre Cakar, and Jörg Hähner
   In Proceedings of the 6th International Conference on Informatics in Control, Automation and Robotics (ICINCO), 2009, pp. 285-290

5. **Self-Organising Interaction Patterns of Homogeneous and Heterogeneous Multi-Agent Populations**
   Emre Cakar and Christian Müller-Schloer

In Proceedings of the 3rd International Conference on Self-Adaptive and Self-Organizing Systems (SASO), 2009, pp. 165-174

6. **Adaptivity and Self-organisation in Organic Computing Systems**
   Hartmut Schmeck, Christian Müller-Schloer, Emre Cakar, Moez Mnif and Urban Richter
   ACM Transactions on Autonomous and Adaptive Systems, Vol. 5, pp. 10:1-10:32, September 2010

7. **Decentralised and Adaptive Collaboration in Multi-Agent Systems**
   Emre Cakar and Christian Müller-Schloer
   In Proceedings of the 9th International Symposium on Parallel and Distributed Computing (ISPDC 2010), July 2010, pp. 195-202

8. **Aspects of Learning in OC Systems**
   Emre Cakar, Nugroho Fredivianus, Jörg Hähner, Jürgen Branke, Christian Müller-Schloer, and Hartmut Schmeck
   In "Organic Computing - A Paradigm Shift for Complex Systems", Christian Müller-Schloer, Hartmut Schmeck, and Theo Ungerer, Eds., incollection 3.1, pp. 237-251. Birkhäuser, Juni 2011.

9. **A Role-based Imitation Algorithm for the Optimisation in Dynamic Fitness Landscapes**
   Emre Cakar, Sven Tomforde and Christian Müller-Schloer
   In IEEE Swarm Intelligence Symposium, SIS 2011, Paris, France, pp. 139-146

Chapter 1

Introduction

1.1 Motivation

Evolution is not only a biological principle, which defines the adaptation of species to their environment, but a universal concept that is omnipresent in everyday life. In all possible areas such as economy [1], computer science [2], medicine [3] or chemistry [4] humankind always strive for perfection by obtaining the maximum benefit with the minimum effort. For example, the objective of economy is to *maximise* the profit and at the same time minimise the costs, while the objective of medicine is to *minimise* the time required to diagnose a certain disease in order to begin immediately with the treatment. If we look at these different areas, we quickly realise that they either try to maximise or minimise different aspects important to the particular area. In terms of mathematics this means that in each case we have an *objective function*, which defines either a minimisation or a maximisation problem. Each solution for the given problem has a particular quality, which is also called *fitness* in terms of Evolutionary Biology [5]. In this context, optimisation, which is the branch of applied mathematics and numerical analysis, is defined as the task of finding the best solution for the given problem from a set of solutions [6]. Since each research area has its own mathematical discipline dealing with it, it is possible to give an abstract representation of the corresponding optimi-

sation task in terms of mathematical functions and use different optimisation techniques from the domain of computer science to find the best solution for the particular problem.

In this thesis, we particularly deal with Organic Computing (OC) systems that can adapt their behaviour and structure to the changes in their operational environment. OC has emerged as a form of biologically-inspired computing [7, 8, 9, 10], and deals with technical systems, which consist of a (usually) large number of elements that interact with each other in order to accomplish a given task. In OC, we investigate different techniques to endow technical systems with life-like properties (e.g., self-organisation, self-configuration and self-optimisation) to give them the capability to adapt their behaviour to the changes in their environment. In this context, the greatest challenge in OC is the search for an adequate system behaviour that provides the best system performance regarding the current conditions of the environment. Moreover, this solution has to be found in limited time, since (1) in many cases a fast reaction is mandatory, and (2) the situation might change, which makes the found solutions obsolete. The task of finding such an adequate system behaviour becomes even more complex with the increasing number of system elements and their behavioral repertoire resulting in large configuration spaces. Thus, an organic system must have an appropriate mechanism to determine the best system configuration for the current environmental conditions in a short period of time. This problem can be regarded as an *optimisation* task, where the system tries to find the *solution* (i.e., the configuration) in the search space (i.e., the configuration space) that produces the *best* (i.e., optimal) system performance.

This thesis concentrates on numerical optimisation in OC systems. Here, we define the optimisation characteristics of OC, and determine the requirements for an optimisation algorithm that must be satisfied for its usage in OC systems. Accordingly, we introduce a new optimisation technique for OC and provide a comprehensive comparison of our technique to well-known state-of-the-art optimisation techniques from the literature. Furthermore, we present the advantages and disadvantages of our approach in different problem settings from the domains of numerical optimisation and OC.

1.2 Problem Statement and Contribution

1.2.1 The two-layer Observer/Controller Architecture

In OC we deal with technical systems that consist of a large number of elements (agents) interacting with each other and also with the environment in order to accomplish a given goal. Typically, these interactions change the environmental conditions requiring an intelligent mechanism to determine the optimal system behaviour at any given point in time. This fact presents two major challenges requiring:

1. A fast and resource-efficient learning mechanism, which facilitates a quick response to the situations that are close to previously encountered *known* situations and

2. An effective optimisation algorithm, which can quickly evaluate different solutions in a (possibly very large) search space in order to find adequate solutions for new previously *unknown* situations.

In this context, the two-layer Observer/Controller architecture for OC systems has been developed (see Fig. 1.1) to react reasonably on changes in the environment in a short period of time [11, 12].

On the lowest layer (Layer 0) of the proposed architecture is the productive system (System under Observation and Control - SuOC). The SuOC may be any kind of decentralised and parameterisable system, which consists of a set of elements possessing certain observable attributes. The parameter selection for the SuOC is implemented on Layer 1 using an eXtended Classifier System (XCS) [13]. Here, an observer component determines the current situation in the SuOC and the controller selects an adequate action for the corresponding situation using an XCS facilitating a quick response to the encountered situation (see requirement 1 above). Layer 2 is triggered each time the observed situation on Layer 0 is not covered by the population of the XCS on Layer 1. In this case, an optimised classifier covering the corresponding situation is created by using a model-based optimisation on Layer 2. Here, an optimisation algorithm evaluates different solutions for the observed situation using a

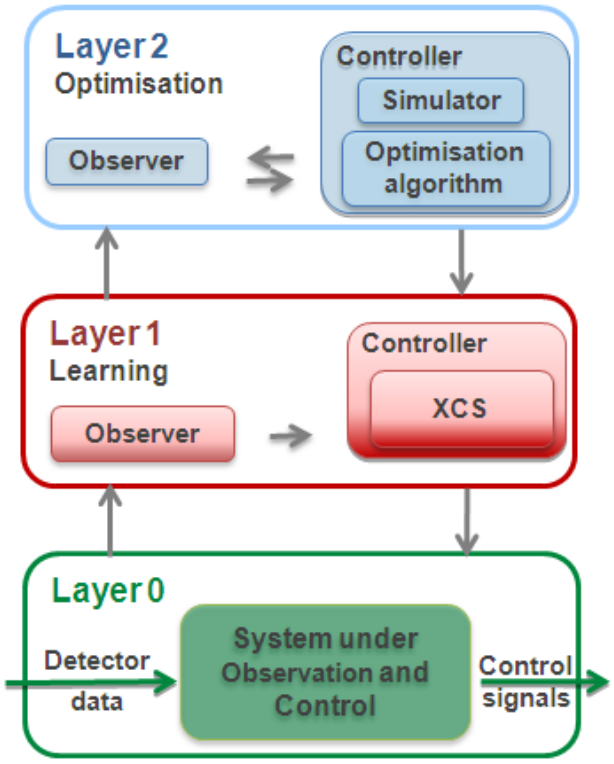


Figure 1.1: The two-layer Observer/Controller architecture.

simulation model of the SuOC. At the end of the optimisation a new classifier is created, which maps the observed situation to the best solution found by the optimisation algorithm facilitating a fast evolution of different solutions for a given situation (see requirement 2 above). In this thesis, we concentrate on Layer 2 of the two-layer Observer/Controller architecture, and investigate the requirements for an optimisation algorithm, which can be used for the optimisation in dynamic environments. Moreover, we introduce a novel optimisation algorithm, the Role-based Imitation algorithm (RBI), for Layer 2 to find high-quality solutions in a short time to react quickly to the changes in the environment.

1.2.2 Optimisation on Layer 2

Each optimisation algorithm must preserve a particular balance between the *exploitation* of already found good solutions and the *exploration* of new (pos-

sibly better) solutions. There are two main approaches in the literature regarding how this balance can be reached. The first approach involves more exploration at the beginning of the optimisation in order to determine good solutions, and switches to more exploitation towards the end of the optimisation in order to make use of best solutions found. For example, the well-known optimisation algorithm Simulated Annealing (SA) [14] uses this approach. The second approach uses a particular optimisation pattern, where exploration and exploitation take place simultaneously according to predefined criteria typically expressed in form of probabilities. Evolutionary Algorithms (EAs) [15], which use biologically-inspired mechanisms like crossover and mutation to refine a set of possible solutions, use this pattern. EAs utilise a set of *individuals* (i.e., abstract entities or abstract agents) to represent possible solutions in the search space. The individuals exchange information about their corresponding fitness values in order to find the best solution in the given fitness landscape. In this case, each individual explores and exploits at the same time according to predefined mutation and crossover probabilities (see Sec. 3.3.1).

Although both approaches are used in the literature to optimise problems from different domains [6], they have some weaknesses, which prevents them to be used effectively in OC systems. In contrast to pure optimisation problems, the optimisation in OC involves real systems (agents), which explore the search space in order to adapt their behaviour to the current conditions of the environment. Since these conditions may change over time, there is no guarantee that a particular solution has always the same quality (i.e., fitness) at every operational stage of the system. In other words, the solution, which produces the best system performance (i.e., optimum), may change over time according to the conditions of the environment. This results in a dynamic optimisation problem, where the two approaches mentioned previously face some difficulties. In this context, the first approach would fail since the corresponding optimisation algorithm switches from more exploration to more exploitation losing the capability to track the moving optimum over time. The weakness of the second approach lies in its optimisation scheme, where each individual explores and exploits at the same time according to predefined probabilities. In this case, the exploration of the search space allows to find better solutions,

while the exploitation results in the acceptance of solutions that are solely in the neighbourhood of the current best solution. Thus, the *greedy* exploitation works against the *creative* exploration and increases the time, which is required to determine the new optimum in dynamic optimisation problems. Hence, a system that uses this kind of optimisation algorithm requires a large amount of time to react to changes in the environment and to adapt its behaviour accordingly. This suggests a clear distinction between the exploring and exploiting individuals to minimise time required to find the optimum.

Based on the requirements discussed in Sec. 1.2.1 and in Sec. 1.2.2, we introduce in this thesis the Role-based Imitation algorithm (RBI), which provides a clear distinction between exploring and exploiting individuals (see Chapter 4). RBI uses a dynamic role-assignment strategy to determine the portion of *purely exploiting* and *purely exploring* individuals in the population, and guarantees the consistent exploration of the search space to track the (moving) optimum at every operational stage of the system. Thus, RBI determines the adequate balance between exploitation and exploration for a given problem adaptively without (1) the need for a mechanism to switch from more exploration to more exploitation over time as in SA, or (2) predefining probabilities for exploration and exploitation as in EA. These properties makes RBI a very suitable candidate to be used on Layer 2 of the two-layer Observer/Controller architecture to effectively track the moving optimum in dynamic environments.

1.3 Outline

This thesis is organised as follows. Chapter 2 presents different aspects of optimisation that are required to define the scope of this thesis. There, a classification of existing optimisation tasks is provided, and the main characteristics of the optimisation in OC systems are discussed. Chapter 3 gives an overview of related work. There is a wide variety of optimisation algorithms, and therefore it is not possible to consider all of them in this thesis. Hence, we have limited our investigations to a particular set of algorithms and use it for comparison purposes. In this context, we provide in Chapter 3 a clas-

sification of the investigated algorithms and present them in detail. Chapter 4 describes our main contribution, the Role-based Imitation algorithm (RBI), in detail, and presents the implementation of RBI for different types of search spaces. Chapter 5 and Chapter 6 provide a comprehensive comparison of RBI to state-of-the-art optimisation algorithms presented in Chapter 3 in case of static and dynamic optimisation problems, respectively. Chapter 7 concludes our work and discusses the resulting future research opportunities.

Chapter 2

Optimisation for OC Systems

In this chapter, we aim at defining the scope of this thesis. In this context, we define the term *fitness landscape*, and explain different types of optimisation tasks, which are common to all optimisation processes, in order to position our work within the current research environment. Furthermore, we extend the existing classification of fitness landscapes from the literature and provide a new view on dynamic optimisation tasks, which is important for OC systems.

2.1 Definition: Fitness Landscape

The term "fitness landscape", which has been introduced into evolutionary biology already in 1932 by Sewall Wright [16], defines one of the most important aspects in optimisation. Principally, a fitness landscape visualises the relationship between the phenotypes and their corresponding reproduction probability in evolutionary biology [17]. The idea of visualising such a relationship has been adopted by computer scientists in order to determine and classify the complexity of a given optimisation problem. In evolutionary optimisation, the term "fitness landscape" defines the set of mappings between all candidate solutions for a given problem (search space) and their fitness values (i.e., qualities). This kind of abstraction provides the simplification that an optimiser can be considered as a population of hikers (individuals or agents) with a

limited view, which try to find the lowest valley or the highest hilltop in the corresponding fitness landscape (see Fig. 2.1).

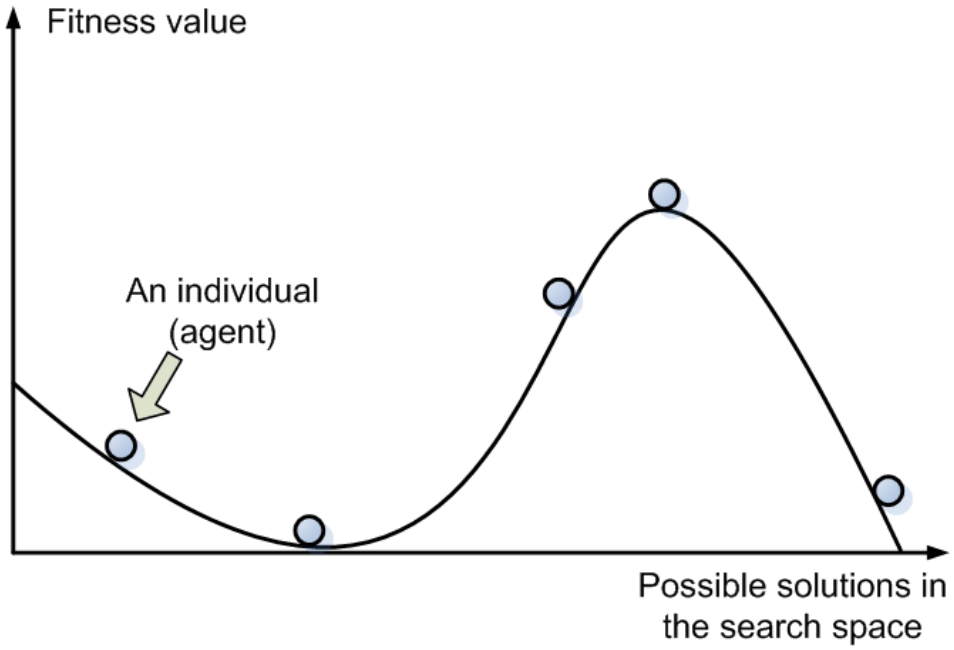


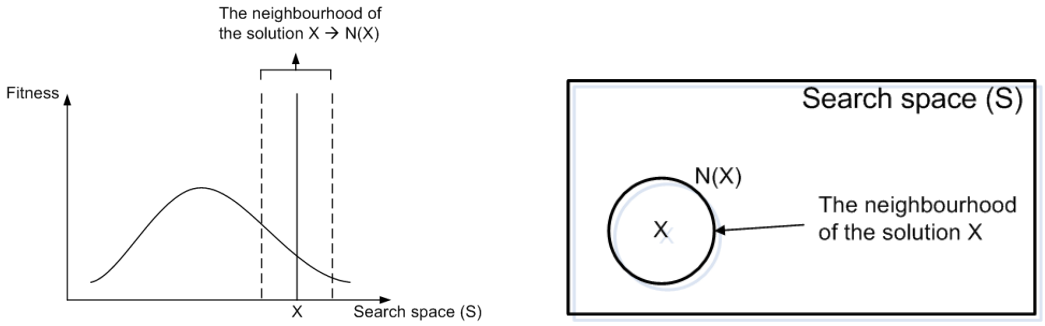
Figure 2.1: A fitness landscape together with the individuals (agents) who search for the lowest/highest point

The lowest valley or the highest hilltop are called the *optimum* in the given fitness landscape depending on the type of the optimisation problem. An optimum typically indicates the maximum (or in case of a minimisation problem the minimum) fitness value in the corresponding fitness landscape. In global optimisation, it is a convention that optimisation problems are always defined as minimisation problems [6]. If the given problem is a maximisation problem, we can minimise its negation.

In many real-world problems we distinguish between the *global* and the *local* optimum. In order to explain these terms please consider the abstract search spaces illustrated in Fig. 2.2.

In Fig. 2.2 X indicates a solution in the search space S and N(X) shows the neighbourhood of X, which is defined as follows:

$$N(X) = Y \in S : d(X, Y) \leq \epsilon \tag{2.1}$$



(a) A 1-dimensional search space (b) A 2-dimensional search space

Figure 2.2: The abstract search space S and a solution X with its neighbourhood N(X)

where $d : S \times S \rightarrow \Re$ is a distance measure and $\epsilon \geq 0$ [18]. Thus, a solution Y is in the ϵ-neighbourhood of X, if it satisfies the condition given in Eq. 2.1. In this context, the term *global* optimum indicates the best fitness value that can be obtained based on all solutions in the search space (global neighbourhood), while the term *local* optimum indicates the best fitness value according to a particular subset of solutions in the search space (local neighborhood). For example, if we have a minimisation problem (e.g., a function optimisation) and the solution X produces the lowest fitness value in comparison to the other solutions in its neighbourhood $N(X)$, then we can call $f(X)$ the local optimum (in this case the minimum). In case of the global optimum, we consider not a specific neighbourhood but the whole search space. If the solution X produces the lowest fitness value in comparison to all other solutions in the search space S, then $f(X)$ is called the global optimum (see Fig. 2.3).

According to the type of the given fitness landscape, it is possible that more than one solution produces the same fitness value for the given problem. Hence, if the fitness landscape is previously unknown, which is mostly the case in global optimisation problems, we typically must assume that there exists more than one optimum.

After having defined the term "Fitness landscape" informally above, we can now consider a more formal definition of it. A fitness landscape L is a triple $L = (\chi, f, d)$ [19], where χ is the set of all candidate solutions (i.e., the search space), f is the evaluation function and d is a distance measure

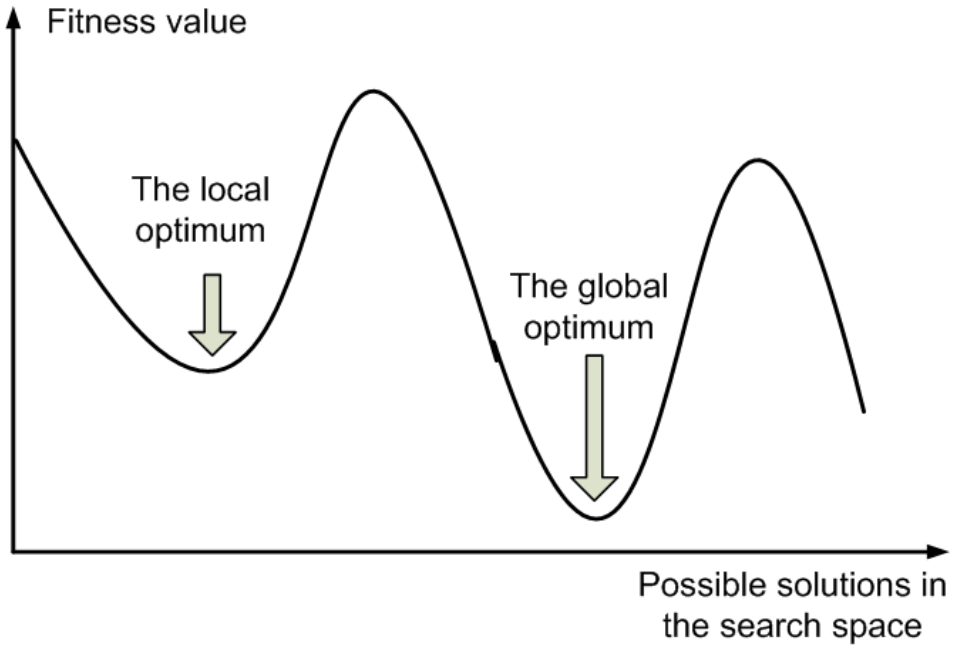


Figure 2.3: The local and global optima.

$d : \chi \times \chi \rightarrow \Re^{+} \cup \infty$ for which it is required that, $\forall$ s, t, u $\in \chi$,

$$\begin{aligned} d(s,t) &\geq 0 \\ d(s,t) = 0 &\Leftrightarrow s = t \\ d(s,u) &\leq d(s,t) + d(t,u) \end{aligned} \tag{2.2}$$

Generally, fitness landscapes are classified according to their form. Fitness landscapes with only one local optimum are called "unimodal", whereas fitness landscapes with more than one local optimum are called "multimodal" [19]. Having this type of classification, it is possible to further differentiate between the landscapes according to the information they provide to an optimisation algorithm. In this context, there are amongst others needle-in-a-haystack fitness landscapes [20], which have no structure to exploit or noisy [21] and deceptive fitness landscapes [22], where unreliable solutions mislead the search of the optimum.

2.2 Fitness Landscapes of OC Systems

In OC, we deal with technical systems that consist of a large number of elements, which continuously interact with each other. These interactions result either in static fitness landscapes, where the mapping between the solutions and their fitness values does not change over time, or in dynamic fitness landscapes, where the solution-fitness mapping is not fixed, which means that it is not guaranteed that a solution S always produces the same fitness value at any point in time ($f_{t_1}(S) \neq f_{t_2}(S)$). In case of dynamic fitness landscapes, the solution-fitness mapping can change according to different conditions. In this context, we classify dynamic fitness landscapes into two groups: (1) Time-varying fitness landscapes and (2) self-referential fitness landscapes. The main difference between a time-varying and a self-referential fitness landscape is that a time-varying one changes only as a function of time regardless of the behaviour of system elements, while the form of a self-referential fitness landscape is influenced by the behaviour of each system element. This classification of the fitness landscapes according to the change of their form over time can be used orthogonally to the standard classification from the literature mentioned in Sec. 2.1. In the following, we present static, time-variant and self-referential fitness landscapes in more detail.

2.2.1 Static fitness landscapes

In a static fitness landscape the mapping between the solutions in the search space χ and their fitness values does not change over time, which can be characterised as follows: $S \in \chi$ and $t_1,\ t_2 \in T,\ t_1 \neq t_2\ ,\ f_{t_1}(S) = f_{t_2}(S)$, where T is an arbitrary time interval from the beginning to the end of the optimisation. This means, f does not depend on time so that the solution S produces the **same** fitness value at any point in time. The optimisation in this type of fitness landscapes is explicitly investigated in the context of function optimisation [23]. In this case, the corresponding optimisation algorithm tries to find the optimum e.g., in a unimodal or a multimodal fitness landscape using the advantage of the fixed solution-fitness mapping.

2.2.2 Time-variant fitness landscapes

As mentioned previously, in case of a time-variant fitness landscape the mapping between the solutions in the search space and their fitness values are not fixed, rather it changes as a result of *outside* conditions. The typical optimisation problems in traffic scenarios are good examples for problems with time-varying fitness landscapes [24]. Particularly, the optimisation of signal plans for traffic light controllers in urban traffic networks presents great challenges in terms of finding the optimum in a fitness landscape, which changes as a function of time. In this context, we consider a large network of traffic light controllers, which adapt their signal plan (behaviour) to constantly changing traffic demands during the day. In this context, each signal plan corresponds to a solution S in the search space, which determines the cycle time[1] and the length of green and red phases in this cycle time for the corresponding intersection. Typically, an optimal signal plan (i.e., a solution S) for a traffic situation in the morning would not provide the same fitness value (e.g., the average delay time per car passing the intersection) during the whole day (e.g., in the midday or at night), since the corresponding traffic densities change over time. In terms of optimisation it means that the solution-fitness mapping changes as a function of time presenting a high level of complexity, which requires sophisticated learning and optimisation algorithms to track the moving optimum. In [24], Tomforde et al. present the *Organic Traffic Control* (OTC) approach, which is based on the generic Observer/Controller architecture discussed in Sec. 1.2.1, to facilitate the self-adaptation of traffic light controllers to time-variant conditions of the environment.

2.2.3 Self-referential fitness landscapes

In contrast to a time-variant fitness landscape, a self-referential fitness landscape changes as a result of *inherent* conditions. In this case, the form of the particular fitness landscape is influenced by the behaviour of the system elements themselves, which makes the optimisation task much harder than in

[1] The cycle time determines how long it takes until the red and green phases for the traffic light controller have been activated and the cycle is restarted.

static and time-varying fitness landscapes. As an example for a self-referential fitness landscape, we can consider again a traffic scenario, where the optimisation does not take place on the traffic light controller level, but rather on the vehicle level. In this context, the fitness landscape of such a traffic scenario can be considered as self-referential, where the optimal behaviour of a system element (e.g., a car) depends on the behaviour of other elements in the system. For example, an optimal route for a car from a point A to another point B may not be optimal after some time, if other cars also decide to take the same route, and this may cause a traffic congestion on this particular route. Thus, the mapping between the solution (the route from A to B) and the corresponding fitness value (the time required to drive from A to B) changes, if other system elements change their behaviour. In OC, we mainly investigate problems with a self-referential fitness landscape [25, 26, 27]. In this context, it is important to distinguish between pure optimisation problems, where the agents (individuals) are solely used to scan a given fitness landscape without influencing it, and the optimisation in OC systems, where the agents represent real systems (like cars in a city), which at the same time explore the fitness landscape and act within the fitness landscape (hereby changing it). Traffic systems are just one example for a self-referential fitness landscape. Another examples are the stock market where "acting" means buying or selling stocks (which of course immediately changes the price of the stock) or the prominent Minority Game [28]. Of course in OC the fitness landscape is usually not explicitly known.

2.3 Optimisation Tasks

Fitness landscapes define only one important aspect of optimisation. In order to provide a deeper understanding of the optimisation theory, we also have to consider the following aspects:

1. The type of the search space (continuous/discrete)
2. The type of the objective function (single-objective/multi-objective)
3. The type of the optimisation (online/offline)

In the following, we present these aspects in more detail.

2.3.1 The Type of the Search Space

In recent years, many efforts have been devoted to developing optimisation algorithms, where the goal is to find the best possible set of parameters (i.e., an optimal solution) according to some given criteria typically expressed as mathematical functions. In this context, an optimisation problem can have a search space, which is either defined over continuous variables (e.g., the optimisation of the Griewank function [29, 23]) or over discrete variables (e.g., the Traveling Salesman Problem - TSP [30, 31]). In each case, we need an adequate distance function (see Section 2.1) in order to define the neighbourhood of solutions in the search space, which in turn determines the form of the corresponding fitness landscape. It is relatively easy to define a distance function for search spaces defined over continuous variables. For this purpose, we basically use the Euclidean metric ($\sqrt{\sum_{n=0}^{n}(X_i - Y_i)^2}$) to determine the distance between two solutions X and Y. The problem of finding an appropriate distance function becomes harder, if the problem has a discrete search space. In this context, different operators may lead to completely different fitness landscapes, which directly affects the optimisation process and the quality of solutions obtained by a particular optimisation algorithm. In order to show this effect, we consider the Bit-flip (BF) and Complementary Crossover (CX) operators [19]. The BF-operator determines the neighbours of a given binary solution by simply flipping one of the bits as follows:

$$BF : 0,1^l \times Z \rightarrow 0,1^l \begin{cases} z_k^{'} = 1 - z_k & \text{if } i = k \\ z_k^{'} = z_k & \text{otherwise} \end{cases} \tag{2.3}$$

where z is a binary string of length l, and i is the parameter specifying the index of the bit to be flipped. For example, the BF-neighbours of the binary string "0000" would be (1000),(0100),(0010),(0001). Accordingly, the well-known Hamming distance can be used to determine the distance between the solutions. In this case, all four solutions (1000),(0100),(0010),(0001) would be in the 1-step neighbourhood of the solution "0000" according to the Hamming

distance. The things get complicated, if we use the CX-operator. The CX-operator determines the neighbours of a binary solution as follows:

$$CX : 0,1^l \times Z \rightarrow 0,1^l \begin{cases} z_k^{'} = 1 - z_k & \text{for } k \geq i \\ z_k^{'} = z_k & \text{otherwise} \end{cases} \tag{2.4}$$

where z is a binary string of length l, and i is the parameter specifying the index of the bit to be flipped and $k = 1, ..., l$. Hence, the CX-neighbourhood of the binary string "0000" now become (1111), (0111), (0011), (0001). In this case, we have a completely different set of solutions in the 1-step neighbourhood of the solution "0000" so that the distance cannot be easily described by using e.g., the Hamming distance.

The example above shows clearly that the optimisation problems with discrete variables require (1) an appropriate operator to determine the neighbourhood of a given solution and (2) an adequate metric to determine the distance between the solutions in the search space.

In order to exemplify the effect of a particular operator on the fitness landscape of a given problem, we consider the *Onemax* problem that is about to maximise the function F defined in Eq. 2.5:

$$F = max(\sum_{i=1}^{l} x_i), x_i \in 0,1 \tag{2.5}$$

As can be seen in Eq. 2.5, the goal in the Onemax problem is to find the binary the string of length l, where the components of the corresponding binary solution consists only of 1's. Based on this problem, the BF-operator we have defined in Eq. 2.3 would sort the possible solutions in the search space so that we have a unimodal fitness landscape, which can be optimised by simply using the standard Hill Climbing algorithm (see Sec. 3.2). The CX-operator, on the other hand, creates a highly multimodal fitness landscape for the Onemax problem, where the number of local optima increases exponentially with the length l of the binary string (i.e., with the size of the search space). In this case, the Hill Climbing algorithm would get stuck in a local optimum without a chance to determine the global one [32]. This example shows clearly that the utilisation of a particular operator determines the form of the fitness

landscape and affects directly the performance of the corresponding optimisation algorithm. Thus, the selection of an adequate operator depends highly on the particular problem, and must be determined based on the underlying semantics of the corresponding problem.

2.3.2 The Type of the Objective Function

Another important aspect in optimisation is the objective function defining the goal to achieve. In this context, we can classify the existing types of objective functions into two groups: (1) Single objective functions and (2) multiple objective functions. In the first case, the optimisation problem is either a maximisation or a minimisation of *only one criterion* of the given problem. For example consider a traffic light controller that tries to minimise the average waiting time in a junction [33] or a firm that tries to maximise its annual profit. In single objective real-world optimisation problems, we mostly deal with fitness landscapes that have many *local* optima (i.e., multimodal fitness landscapes) requiring the corresponding optimisation algorithm to have adequate techniques to avoid them.

The second type of objective functions - multiple objective functions - involves not only one criterion in optimisation, rather there are at *least two criteria*, which should be optimised at the same time [34]. Thus, this kind of optimisation is also called multi-criteria optimisation or vector optimisation [35]. In this context, the objective functions are elements of an objective vector, and the task is to find a solution, which satisfies all objective functions (i.e., criteria). In this case, different objective functions may be in conflict with each other so that it would be impossible to increase the fitness of one objective function without decreasing the fitness of at least one another objective function. Hence, a solution X in the search space S is called *Pareto optimal*, if it satisfies this condition[2]. More formally, Pareto optimality can be defined using the notion of *domination*. In this context, a solution s_1 dominates another solution s_2 (i.e., $s_1 \vdash s_2$), if s_1 is better than s_2 in at least one objective function and not worse with respect to all other objective functions

[2]This fact was formulated by Vilfredo Pareto in the XIX century, and this concept is associated with his name.

[6]. If F is the set of all objective functions f_i, we can define $s_1 \vdash s_2$ as shown in Eq. 2.6.

$$\begin{aligned} s_1 \vdash s_2 \Leftrightarrow \forall i : 0 < i \leq n \Rightarrow \omega_i f_i(s_1) \leq \omega_i f_i(s_2) \wedge \\ \exists j : 0 < j \leq n \Rightarrow \omega_i f_i(s_1) < \omega_i f_i(s_2) \\ \omega_i = \begin{cases} 1 & \text{if } f_i \text{ should be minimised} \\ -1 & \text{if } f_i \text{ should be maximised} \end{cases} \end{aligned} \tag{2.6}$$

Based on the definition of domination from Eq. 2.6, a solution s_1 is Pareto optimal, if it is not dominated by any other solution in the search space S (see Eq. 2.7)[6].

$$s_1 \in S \Leftrightarrow \nexists s_i \in S : s_i \vdash s_1 \tag{2.7}$$

The concept of Pareto optimality has a wide application area and is also apparent in multiagent systems. A multiagent system forms a particular type of distributed system, which is composed of multiple interacting computing autonomous elements called agents [36]. Since the agents are autonomous, they can take actions on their own (either according to a predefined behaviour repertoire or they learn new actions) in order to accomplish their goals. At this point, there are two possibilities to define goals for the agents: (1) The goal of each agent is a small part of a given global goal so that the agents *collaborate* with each other to accomplish the global goal [37, 38] or (2) the agents have different conflicting goals and they *compete* with each other to accomplish their own goals [39]. Pareto optimal solutions belong mostly to the second case, where agents have conflicting goals and compete with each other. Fig. 2.4 shows the Pareto optimal solutions for two competing agents i and j [36].

In Fig. 2.4 the solutions on the line from B to C are Pareto optimal, since we cannot increase the fitness of one agent without decreasing the fitness of the other one [36].

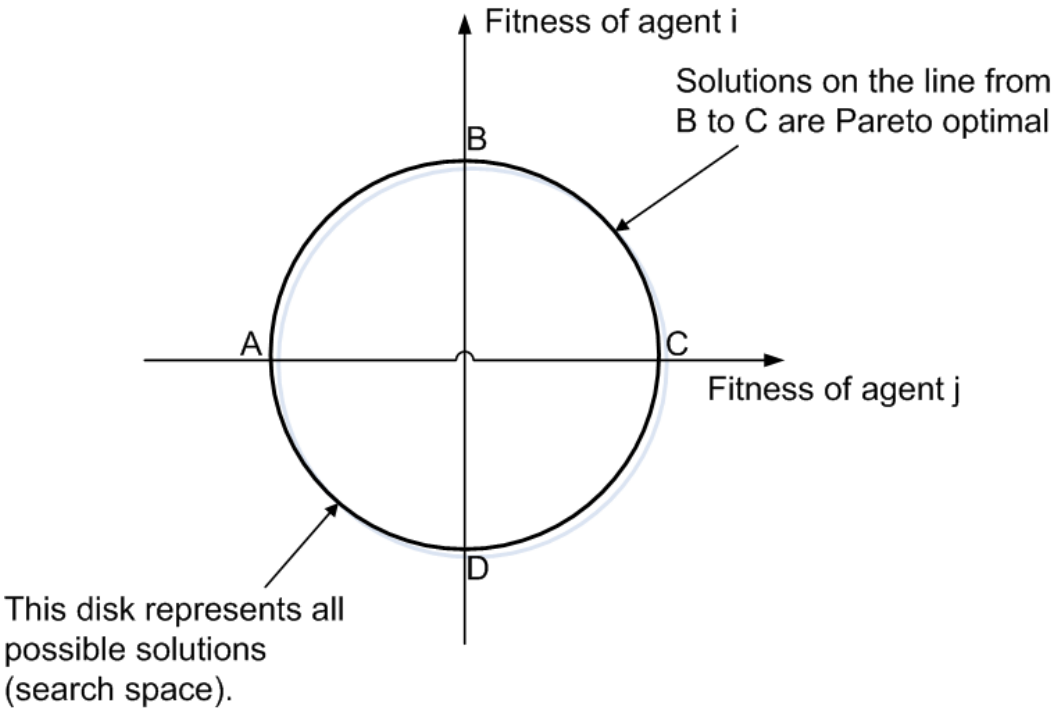


Figure 2.4: The abstract representation of Pareto optimal solutions for a system consisting of two agents i and j

2.3.3 The Type of the Optimisation

Generally, in optimisation we consider two major criteria to identify the performance of the corresponding optimisation algorithm: (1) The quality of solutions obtained by the algorithm and (2) the time required to find adequate solutions. The second criterion can be used to classify the type of the optimisation. In this context, we distinguish between two main use cases: (1) Online (runtime) optimisation and (2) offline (design time) optimisation. The first type of optimisation occurs in systems that are in operational state. In this case, the optimisation process must be carried out repetitively in **short** periods of time to provide an acceptable solution e.g., to be able to react to the changes in the environment, and guarantee that the corresponding system can continue working even in the presence of fluctuations [40]. In this context, we consider the typical optimisation cycle presented in Fig. 2.5.

Fig. 2.5 shows two major steps we observe in each optimisation process.

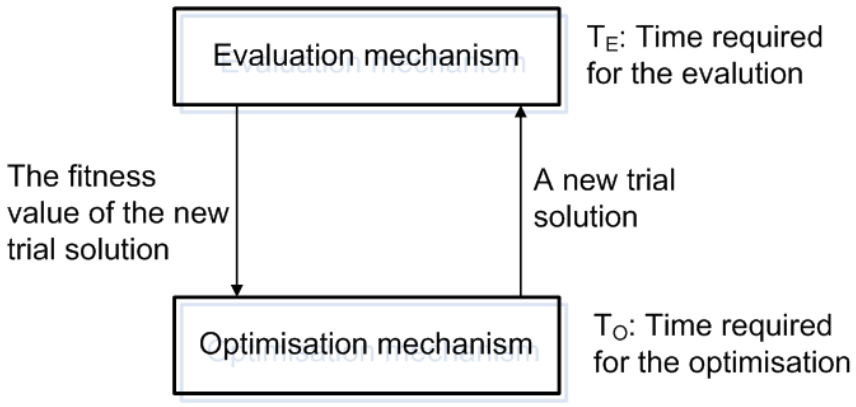


Figure 2.5: The abstract representation of an optimisation process.

Generally, the optimisation mechanism (optimiser) produces according to its internal scheme a new trial solution and sends it to the corresponding evaluation mechanism. The time required for this part of the optimisation process is denoted as T_O. After having sent the trial solution, the evaluation mechanism calculates the fitness value of the corresponding solution, and sends this value back to the optimiser. The time required for the evaluation part of the optimisation process is denoted as T_E. In many real-world problems, T_E is much larger then T_O so that we can omit T_O in calculating the time required to optimise a given problem [24]. Thus, the maximum amount of time required for the optimisation process is always strongly correlated to the number of (function) evaluations that are required to find an adequate solution. This fact becomes more apparent especially in cases where the evaluation mechanism is not a simple function, but rather a complex simulation process as proposed in [33]. Thus, an optimisation algorithm that is used to solve an online optimisation task must be able to create an adequate balance between the exploration of the fitness landscape and the exploitation of the learnt knowledge about the fitness landscape to minimise the number of function evaluations required to find the optimal solution.

The second type of optimisation problems consists of design time optimisation tasks, which are basically carried out only once in a long time. In this context, the time is not as important as in the runtime optimisation and can take even days until the corresponding algorithm produces the optimal

solution. This type of optimisation takes place for example in data mining [41] in order to correctly extract previously unknown and potentially useful information from large databases.

2.4 Classification and Scope

In this chapter, we have presented four important aspects of optimisation in order to define the scope of this thesis. These aspects are (1) the fitness landscape, which can be either static or dynamic, (2) the type of the search space, which can be defined over continuous or discrete variables, (3) the type of the objective function, which is either single-objective or multi-objective, and (4) the type of the optimisation, which takes place either in runtime or in design time.

The main subject of this thesis is the runtime optimisation of problems defined over continuous variables with the Role-based Imitation algorithm (RBI) in static and self-referential fitness landscapes (see Sec. 4.2). For each type of fitness landscape (static and self-referential), we investigate three main criteria:

1. **Solution quality in noiseless environments**
   Each optimisation algorithm uses a different technique to determine a particular balance between the exploitation of already found good solutions and the exploration of new solutions. In order to determine whether a given algorithm can find an adequate balance of exploration and exploitation for a particular problem, it is required to investigate the performance of the corresponding algorithm with regard to typical difficulties arising from the form of the given fitness landscape. This investigation provides us with information on the investigated optimisation algorithms (and on their exploitation/exploration behaviour) to answer the question about to what extent they are capable of coping with generic aspects of fitness landscapes like multimodality and high dimensionality. Thus, this investigation is of more theoretical nature, since it only deals with different types of fitness landscapes without concerning *noise*, which is the most important aspect of real technical systems.

2. **Solution quality in noisy environments**
   Generally, the optimisation problems that arise in the context of real technical systems involve noise, which may occur due to different reasons such as defect sensors, inaccurate sensor values, different internal or external disturbances and a continuously changing environment. Thus, an optimisation algorithm that is used in real-world applications must not only cope with the difficulties arising from the form of the fitness landscape, but also with the existing noise in the environment. Thus, the investigation of noisy environments is of more practical nature and provides answers for the questions about how a particular optimisation algorithm can provide a consistent exploration of the fitness landscape, and what a suitable exploitation scheme looks like, which can alleviate the negative effect of the noise.

3. **Convergence speed**
   The solution quality is one important criterion for the comparison of different optimisation algorithms. Another important criterion is the convergence speed, which defines the time that the corresponding optimisation algorithm requires to produce a certain system performance. Since we deal with real technical systems in OC, it is important to find good solutions in an acceptable amount of time. This aspect becomes more clear in case of real-time OC systems, where we have specific time limits to achieve a given goal. In this case, we cannot afford to execute an optimisation algorithm beyond these time limits to obtain a good solution, rather such solutions must be found as fast as possible within the given time limits to keep the system in an operational state, where it reaches the optimal performance.

As mentioned above, the main subject of this thesis is the optimisation of problems defined over continuous variables with RBI. However, it is also possible to apply the main idea of RBI to problems defined over discrete variables (see Sec. 4.3). In order to show this applicability we extend our investigations to the area of combinatorial optimisation problems and test the performance of (discrete) RBI using different instances of the Traveling Salesman Problem

(TSP) [42]. We emphasise here that our purpose is not to provide a detailed study of RBI in different types of discrete environments, but rather to show how the basic principles of RBI can be transferred into a discrete search space. Thus, in this case we do not concern a complete set of test cases including the investigation of noisy environments and self-referential fitness landscapes, but study the behaviour of RBI using the TSP in its original form, which is static and noiseless.

Based on the aspects discussed above, we summarise our evaluation strategy for RBI as shown in Fig. 2.6.

	Algorithm 1	Algorithm 2	...
Applicable in			
discrete search spaces			
continous search spaces			
Continous search spaces			
Static fitness landscapes			
Solution quality in noiseless env.			
Solution quality in noisy env.			
Convergence speed			
Self-referential fitness landscapes			
Solution quality in noiseless env.			
Solution quality in noisy env.			
Convergence speed			
Discrete search spaces: TSP			
Static fitness landscapes			
Solution quality in noiseless env.			
Convergence speed			

Figure 2.6: The evaluation strategy used to compare different optimisation algorithms.

In the next chapter, we present the state-of-the-art optimisation algorithms, which we use in our investigations.

Chapter 3

State of the Art: Moving towards the Optimum

In this chapter, we discuss the optimisation algorithms, which we use in our investigations, in detail. There is a wide variety of optimisation algorithms, and therefore it is not possible to consider all of them in this thesis. Hence, we limit our investigations to a particular set of state-of-the-art optimisation algorithms for producing experimental results. Basically, all optimisation algorithms have some similarities to and differences from each other so that a classification of them must be given in order to provide an appropriate overview. In the following, we give a classification of the optimisation algorithms we investigate in this thesis and present them in more detail.

3.1 Classification of Optimisation Algorithms

Existing optimisation algorithms can be classified according to the search strategy they utilise to determine the optimum in the given fitness landscape. In this context, an optimisation algorithm is either *deterministic* or *probabilistic*. A deterministic optimisation algorithm does not use randomness to determine what actions to take in the optimisation process. Hence, such algorithms are used in cases where the mapping between the solution and its fitness value is

fixed (static fitness landscapes) and dimensionality of the problem is low. In such cases, it is possible to enumerate the search space and explore it for example using a Generalised Pattern Search algorithm (GPS) [43]. On the other hand, the probabilistic optimisation algorithms are used to optimise more complex problems, where the search space is large and/or its dimensionality is high. The probabilistic (stochastic) approach incorporates random elements to determine the exploration and exploitation behaviour of a particular optimisation algorithm [44] or even to decide whether a new trial solution should be accepted or not [14]. The use of the randomness has principally two main advantages: (1) It makes the optimisation algorithm less sensitive to modeling errors and (2) it decreases the runtime of the optimisation process, since the search space does not need to be enumerated.

In this thesis, we concentrate on the stochastic optimisation algorithms, since the problems we are concerned with have both a (very) large search space and a high dimensionality. In this context, we classify the existing stochastic algorithms into two groups: (1) Population-based and (2) trajectory-based optimisation algorithms. A population-based optimisation algorithm utilises individuals (agents), which search for the optimum in a given fitness landscape by exchanging information about their solutions and the corresponding fitness values. Thus, the optimisation is carried out *collectively* by a population of agents. A trajectory-based optimisation algorithm, on the other hand, considers only the *form* of the corresponding fitness landscape in order to examine new trial solutions (exploration) or to use the available information about the fitness landscape to get closer to a (possibly local) optimum (exploitation). Fig. 3.1 shows the classification of trajectory-based and population-based optimisation algorithms.

There is a large number of trajectory-based and population-based optimisation algorithms so that it is not possible to consider all of them in this thesis [6]. Thus, we limit our investigations to the most prominent and well-studied members from each class of optimisation algorithms, which are shown in white boxes in Fig. 3.1. The algorithms in grey boxes are just further members of each class of optimisation algorithms. Please notice that Fig. 3.1 does not give the complete list of all existing stochastic optimisation algorithms, rather

it shows only a subset of them in order to demonstrate the basic semantics we used for classification purposes.

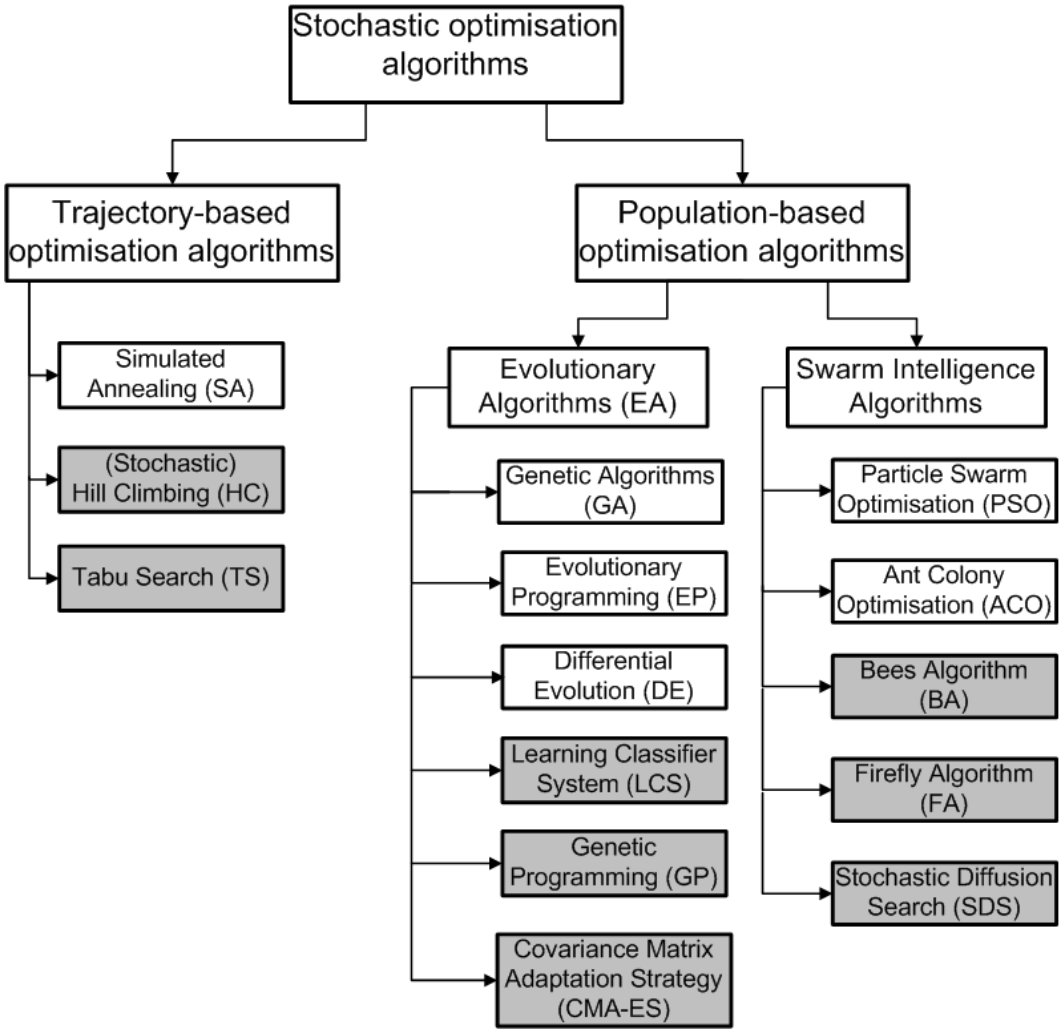


Figure 3.1: The classification of stochastic optimisation algorithms. Only the algorithms in white boxes are investigated in this thesis. For a more detailed classification of existing optimisation algorithms please refer to [6].

In the following we present the algorithms, which we use for investigation and comparison purposes in this thesis, in more detail.

3.2 Trajectory-based Optimisation Algorithms

Trajectory-based optimisation algorithms start with a random initial solution and refine it at each step by replacing the current solution with another (often better) solution found in the neighbourhood. The most prominent member of this class of algorithms is the well-known Hill Climbing (HC) algorithm. The HC algorithm is actually deterministic and does not utilise random elements in the optimisation. However, HC is a very generic optimisation model, and thus serves as a basic template for many stochastic trajectory-based optimisation algorithms like Stochastic Hill Climbing [18], Tabu Search [45, 46] or Simulated Annealing [14]. The HC algorithm starts with a random solution, and at each optimisation step it searches the neighbourhood of the current solution for better solutions. The current solution is replaced only if the new solution provides a better fitness value than the current one (see Procedure 1).

Procedure 1 The Hill Climbing procedure

```
1:  BEGIN
2:  define MAX, bestSolution
3:  V_current = random()          ▷ Select a random solution in the search space
4:  bestFitness = eval(V_current)
5:  t = 0
6:  while t < MAX do
7:      select V_new ∈ N(V_current)      ▷ V_new is in the neighbourhood of V_current
8:      if eval(V_new) > bestFitness then
9:          bestFitness = eval(V_new)
10:         bestSolution = V_new
11:     end if
12:     t = t + 1
13: end while
14: V_current = bestSolution
15: END
```

As shown in Procedure 1, HC starts with a random solution (line 3). Until the termination criterion is satisfied (line 6), it selects a trial solution V_{new} from the neighbourhood of the current solution $V_{current}$ (line 7), and accepts V_{new} only if its fitness value is better than the fitness value of $V_{current}$ (lines 8-11). The HC procedure can be improved in that the Procedure 1 is carried

out not only for a single random solution $V_{current}$, but for a specific set of solutions. That means, we can perform HC on different parts of the search space (parallel or sequential) and then combine all results. This improvement of the standard HC algorithm is called Iterated Hill Climbing [18], which is also a deterministic algorithm like the standard HC. Generally, iterated HC provides better results than the standard HC algorithm in the majority of the optimisation problems. Nevertheless, the success of HC and Iterated HC is limited, since both algorithms neglect exploring particular parts of the search space in that they never accept solutions that are worse than the current one. This is basically a problem, if we optimise in non-linear multimodal fitness landscapes, since the *greedy* optimisation algorithms like HC get stuck in a local optimum, and cannot find the global one in such fitness landscapes. In this case, the HC algorithm can be extended by random elements to accept new solutions even if they have lower fitness values than the current one in order to cope with this problem. In the next section, we present the Simulated Annealing algorithm, which is basically a kind of stochastic HC approach.

3.2.1 Simulated Annealing

Simulated Annealing (SA) has been proposed by Scott Kirkpatrick, C. Daniel Gelatt and Mario P. Vecchi in 1983 [14]. SA is inspired by the annealing process of metals, which is used to alter their properties like hardness. Metal crystals with small defects or dislocations weaken the structure of the metal. By properly heating and then cooling the metal it is possible to destroy or move these dislocations. A high temperature increases the energy of the ions so that they can move to proper positions avoiding getting stuck in meta-stable (local optimal) states. Afterwards, the temperature is decreased slowly and the metal approaches its equilibrium (optimal) state [6]. SA incorporates this idea into the optimisation process. In this context, the optimisation is initialised with a high temperature to avoid local optima, i.e., at the beginning of the optimisation there is a high probability that solutions with low fitness values are accepted by SA. The temperature is decreased over time so that the probability to accept bad solutions decreases, too. At the end of the

optimisation, the temperature is so low that the final stages of SA resemble an ordinary hill-climbing algorithm (see Procedure 2).

Procedure 2 The Simulated Annealing procedure

1: $BEGIN$
2: define MAX
3: define T ▷ Initialise the temperature T
4: $best = random(V_{current})$ ▷ Select a random solution in the search space
5: **while** $t < MAX$ **do**
6: select $V_{new} \in N(V_{current})$ ▷ V_{new} is in the neighbourhood of $V_{current}$
7: **if** $eval(V_{new}) > eval(V_{current})$ **then**
8: $best = V_{new}$
9: **else**if $random[0,1) < e^{\frac{eval(V_{new})-eval(V_{curent})}{T}}$
10: $best = V_{new}$
11: **end if**
12: $t = t + 1$
13: $T = decrease(T)$
14: **end while**
15: END

As shown in Procedure 2, SA is basically very similar to the standard HC algorithm. SA accepts a solution if it is better than the current one (line 7) or the condition $random[0,1) < e^{\frac{eval(v_n)-eval(v_c)}{T}}$ holds (line 9). The value of $e^{\frac{eval(v_n)-eval(v_c)}{T}}$ decreases with the decreasing temperature (T) so that the probability to accept bad solutions decreases. The idea of incorporating the annealing process into the optimisation is very promising, but there are some important questions to be answered:

1. What termination criterion should be used (line 2 in Procedure 2)?
2. How can we initialise the temperature (line 3 in Procedure 2)?
3. How can we calculate the temperature for the next optimisation step (line 13 in Procedure 2)?

There is not a generic answer for the questions above. This is actually an important issue regarding all optimisation algorithms so that each of them must be fine-tuned according to the characteristics of the particular problem [47].

SA can be applied to both discrete [48, 49] and continuous search spaces [50, 51]. Thus, it is applicable to a wide variety of problems in various domains such as machine learning [52], networking [53] or image processing [54, 55].

3.3 Population-based Optimisation Algorithms

A population-based optimisation algorithm utilises a set of individuals (or agents in the sense of Organic Computing), which exchange information about the solutions and the corresponding fitness values with each other in order to find the optimum in the given fitness landscape. How the exchanged information is used depends on the particular algorithm. However, there are two generic aspects, which concern all population-based optimisation algorithms. These are (1) the *diversity* and (2) the *similarity* of solutions represented by the individuals. In this context, a population-based optimisation algorithm must maintain a set of diverse solution candidates (at least at the initial stages of the optimisation) in order to avoid a *premature convergence* to a local optimum [56, 57, 58, 59], while the similarity of solutions is desired in order to fine-tune the existing solutions towards the global optimum. To achieve these two goals each population involves two types of individuals: (1) Individuals that *explore* the fitness landscape and (2) individuals that *exploit* the existing information about the fitness landscape. Each population-based optimisation algorithm uses a different method in order maintain a good balance between the exploring and the exploiting individuals. In this context, we investigate two main classes of population-based optimisation algorithms: (1) Evolutionary Algorithms (EA) and (2) Swarm Intelligence (SI) algorithms. In the following, we present EA and SI algorithms in more detail.

3.3.1 Evolutionary Algorithms

Evolutionary Algorithms (EA) utilise biologically-inspired operators like crossover and mutation in order to refine a set of possible solutions iteratively [15, 60]. The following steps are carried out by an EA to refine the solutions: (1) After creating an initial population of random solutions, an EA first determines the

values of the objective function(s) for each solution candidate. The optimisation task may expose different goals each with its own objective function. These objective values are then used to calculate the fitness value for each solution candidate. (2) Afterwards, EA selects the fittest individuals from the population according to their fitness values for reproduction of new offspring solutions using crossover and mutation. (3) Based on these new candidates, EA checks whether a termination criterion (e.g., a maximum number of function evaluations) is met or not. Accordingly, it either returns the best solution found so far or repeats the steps from 1 to 3.

A *genome* in an EA is an abstraction of the search space and defines the set of all possible solutions. The elements of the genome (search space) are called *genotypes* (chromosomes). An EA aims at finding the genotypes that provide the best fitness values using different solution candidates called *phenotypes* (individuals). A phenotype is an instance of a genotype formed by the corresponding genotype-phenotype mapping. Using the notion of phenotypes, EA facilitates a goal-driven search for the optimum in the search space.

EAs can be applied to search spaces defined over discrete and continuous variables. Thus, the crossover and mutation operations used to create new offspring solutions look different in problems with different types of search spaces. In this context, EAs are used in a wide range of application domains such as function optimisation [61], Constraint Satisfaction Problems (CSP) [62, 63], data mining [64], combinatorial optimisation [65] and telecommunication [66].

Genetic Algorithm

Genetic Algorithms (GAs) belong to the class of EAs and consist of evaluation, selection and reproduction steps, where the genotypes in the corresponding search space are encoded as arrays of elementary types such as binary strings or decimal numbers. The typical GA cycle is shown in Procedure 3.

An individual (phenotype) in a GA has (1) a specific location (genotype) in the search space and (2) a fitness value, which represents the quality of the corresponding solution. Thus, GA creates at the beginning of the optimisation an initial population of random individuals and assigns to each individual a

Procedure 3 The cycle of GA

```
1: define MAX                                          ▷ Halting criteria
2: define t = 0                                        ▷ Increased in each iteration
3: define NP                                           ▷ The population size
4: define POP                                          ▷ The population
5: define S    ▷ The selected individuals for the mutation and recombination
6: define OFFSPRING                                    ▷ The offspring individuals
7:
8: BEGIN
9: POP = createPopulation(NP)
10: assignFitness(POP)
11: while t < MAX do
12:     S = select(POP)
13:     OFFSPRING = createOffspring(S)
14:     assignFitness(OFFSPRING)
15:     POP ← OFFSPRING
16:     t = t + 1
17: end while
18: END
```

genotype to specify its location in the search space (line 9). Afterwards, the corresponding fitness values are assigned to the individuals in order to define the quality of solutions represented by them (line 10). The fitness values are used to select (higher quality) individuals, which are going to be put into the *mating pool* (line 12). The individuals in the mating pool are used for mutation and recombination. There are different methods used for the selection process. The most prominent selection methods are amongst others (1) the Roulette-wheel selection, where the selection probability of each individual is proportional to its fitness value [67], (2) the Truncation selection, where individuals are sorted according to their fitness values and only the best individuals are selected for the mating pool using a specific *truncation threshold* parameter [68] and (3) the Tournament selection, where the best individual is chosen as a parent after a comparison of fitness values of individuals in a randomly selected (sub-) population. The size of the sub-population is determined using the parameter *tournament size* [68]. Each selection algorithm must be adapted to the particular problem, since it strongly affects the performance of

GA [69, 70].

After having selected the individuals for the mating pool, the corresponding mutation and recombination (crossover) operators are applied to the parent individuals in order to create new offspring individuals (line 13). Since GAs can be applied to search spaces defined over discrete and continuous variables, the implementation of the corresponding mutation and crossover operators varies in problem settings with different types of search spaces. In this context, a real-valued GA, which for example optimises continuous functions, can use the arithmetic crossover and Gaussian mutation operators [23], while a discrete GA, which for example optimises the Boolean Satisfiability Problem (SAT) [18], can use the single-point (SPX) or multi-point (MPX) crossover operators together with the single-gene or multi-gene mutation operator [71, 72]. Finally, the offspring individuals are inserted into the population according to the corresponding insertion scheme (line 15) [73].

GAs are used in different domains such as function optimisation [23, 74], scheduling [75], combinatorial optimisation [76], data mining [77] and medicine [78].

Evolutionary Programming

Evolutionary Programming (EP) has been proposed by Fogel in 1964 in his PhD thesis "On the organization of intellect" [79]. Basically, EP and GA are very similar to each other. However, there is an important aspect in EP, which can be used to distinguish EP from GA. A typical GA assumes to have individuals of the same species, which represent different solutions in the search space. EP, on the other hand, assumes to have different species instead of individuals as solution candidates. Thus, EP differs from GA in that it does not make use of crossover, but only *mutation* in producing offspring [80]. Procedure 4 shows the typical EP cycle.

According to Procedure 4, EP creates a random population of size NP at the beginning of the optimisation (line 8), and determines the fitness values of the solution candidates (line 9). Each solution candidate in the population is subject to mutation (line 11). How the mutation is implemented depends on the search space (discrete/continuous) of the problem and should be deter-

Procedure 4 The cycle of EP

```
1: define MAX                                   ▷ Halting criteria
2: define t = 0                                 ▷ Increased in each iteration
3: define NP                                    ▷ The population size
4: define POP                                   ▷ The population
5: define OFFSPRING                             ▷ The offspring individuals
6:
7: BEGIN
8: POP = createPopulation(NP)
9: assignFitness(POP)
10: while t < MAX do
11:     OFFSPRING = mutate(POP)
12:     assignFitness(OFFSPRING)
13:     POP ← OFFSPRING
14:     pairwiseCompare(POP)
15:     refresh(POP)
16:     t = t + 1
17: end while
18: END
```

mined accordingly. After having created the offspring using the corresponding mutation operator, EP assigns fitness values to them according to the objective function (line 12), and puts them into the population (line 13). Afterwards, pairwise comparisons are conducted over all solution candidates in the population (line 14). For each solution candidate, a specific number of opponents are selected from the population with equal probability. After each comparison, a solution candidate receives a "win", if it has a higher fitness value than its opponent. Finally, the solution candidates with the least "wins" are deleted from the population so that the population size is again NP (line 15) [80].

EP is used in different application domains such as machine learning [81], artificial intelligence [82] and combinatorial optimisation [83].

Differential Evolution

Differential Evolution (DE) is a population-based optimisation algorithm, which has been proposed by Storn and Price in 1995 [84]. DE is used to optimise problems defined over continuous spaces (e.g., multidimensional continuous

functions) and has been invented to solve the Chebyshev polynomial fitting problem [85]. Each individual in DE is equipped with an n-dimensional real-valued parameter vector (this vector corresponds to a genotype in the genome), which represents a solution for the given problem. The main idea behind DE is that it generates new parameter vectors by adding a weighted difference vector between two population members to the parameter vector of a third one. The creation of a new offspring population member looks as follows:

Procedure 5 The DE procedure for creating offspring

1: define DIM ▷ The dimensions of the problem
2: define CR ▷ The crossover probability
3: define F ▷ The scaling factor
4: define POP ▷ The population
5:
6: $BEGIN$
7: $L = U(0, DIM]$
8: $R1 = select(POP)$ ▷ The first individual from the population
9: $R2 = select(POP)$ ▷ The second individual from the population
10: $R3 = select(POP)$ ▷ The third individual from the population
11: $temp = X_R$ ▷ Save the current solution of the individual R
12: **while** $(U[0, 1) < CR$ AND $L < DIM)$ **do**
13: $\quad X_R^L = X_{R1}^L + F(X_{R2}^L - X_{R3}^L)$
14: $\quad L = L + 1$
15: **end while**
16: **if** $eval(X_R) < eval(temp)$ **then**
17: $\quad X_R = temp$ ▷ Reset X_R if the previous solution was better
18: **end if**
19: END

In Procedure 5 the variable DIM denotes the number of dimensions of the given problem. Thus, each individual has a real-valued parameter vector of size DIM. In each iteration, an individual optimises a particular number of parameters (genes) in its vector (genotype). For this purpose, the variable L is determined randomly whose value is between 0 and DIM (line 7). After that, three individuals $R1$, $R2$ and $R3$ are selected randomly (lines 8-9-10) to carry out the crossover operation. Before continuing with the crossover, the individual saves its current solution to the variable *temp* (line 11). In order to control the crossover, DE uses the crossover probability (CR) and $U[0, 1)$,

which generates a uniformly distributed random number between 0 and 1. The crossover is carried out as long as $L < DIM$ and $U[0,1) < CR$ (line 12) by adding a weighted difference vector between $R2$ and $R3$ to the parameter vector of a $R1$ for each parameter (gene) with the index L (line 13). After the crossover has been finished, the individual compares the fitness value of the new solution to the fitness value of the previous solution (line 16). The previous solution is restored if the new solution has a lower fitness value (line 17).

DE is used in various application domains such as function optimisation [86], circuit design [87], chemical engineering [88], biology [89] and control engineering [90].

3.3.2 Swarm Intelligence Algorithms

Swarm Intelligence (SI) defines the decentralised and collective behaviour of systems, which consist of a large number of elements interacting with each other. The most important aspect in SI is that each member of a swarm has a limited view and a simple behavioural repertoire so that it is not able to show *intelligent* behaviour on its own. Thus, we do not observe a local intelligence in such systems, rather the intelligence emerges trough the interactions between the members of the swarm, and therefore it is a global property of the whole system [91]. This kind of *collective* behaviour makes it for the swarm possible to solve complex problems in an adaptive and self-organised way using solutions that are emergent rather than predefined [92].

SI has various application domains. These domains can be classified into two groups: (1) Applications with embodied agents that use SI principles to accomplish real-world problems [93, 94, 38, 95, 96, 97, 98] and (2) applications with abstract (optimising) agents that use SI principles to collectively find the optimum in a given fitness landscape [29, 99, 44, 30, 100, 101]. In this thesis, we concentrate on SI-based optimisation, and present two prominent members of this class of algorithms.

Particle Swarm Optimisation

Particle Swarm Optimisation (PSO) is a population-based optimisation algorithm developed by Kennedy and Eberhart in 1995 [44, 102]. PSO is inspired by the behaviour of bird flocking, where individuals spread in the environment to look for food and move around independently. Each individual in PSO has a specific neighbourhood to which it communicates the current state of its search. If an individual achieves to find food, it sends the corresponding information to other individuals in its neighborhood so that these individuals are also propelled towards to food. Since each individual has randomness in its movements, the individuals do not take a deterministic path while approaching to the intended position, which in turn increases the probability to find more advantageous positions. This idea is applied to optimisation problems using artificial individuals (particles). In this case, each particle represents a solution in an n-dimensional search space, and has a *position*, a *velocity* and a specific *neighbourhood* of other particles to exchange information about the current state of its search. In PSO, the positions (i.e., the solutions represented by the particles) are encoded as real-valued parameter vectors[1].

At each optimisation step, a particle knows its best position and the best position found so far in its neighbourhood, and is propelled towards these positions. Here, a particle determines its velocity vector using the following formula 3.1:

$$\begin{aligned} V^i_{new} = \omega * V^i + \varphi_1 * U_1[0,1) * (P_{id} - X_{id}) \\ +\varphi_2 * U_2[0,1) * (P_{gd} - X_{id}) \end{aligned} \tag{3.1}$$

where V^i stands for the current velocity of the particle, ω for the inertia weight, X_{id} for the current position of the particle, P_{id} for the personal best position of the particle, and P_{gd} for the best position found so far in its neighbourhood. φ_1 and φ_2 are the acceleration coefficients, while $U_1[0,1)$ and $U_2[0,1)$ are uniformly distributed random numbers generated between 0 and 1. After having calculated the velocity vector, a particle determines its new

[1]The position of a particle corresponds to a genotype in the EA terminology, while the velocity does not have a counterpart in EA.

position using the following formula 3.2:

$$X^i = X^i + V^i_{new} \tag{3.2}$$

Determining the optimal set of parameters for PSO is a complex task, since the optimal configuration depends on the particular problem (i.e., the fitness landscape) to optimise. However, there are two approaches to determine the "standard" parameter set for PSO. In the first approach, the particles in PSO are initialised with high velocities using a large ω value to be able to explore the fitness landscape at the beginning of the optimisation. The velocities are then decreased over time so that the particles switch from more exploration to more exploitation. Eberhart et al. have used this technique to optimise multidimensional functions, where ω is set to 0.9 at the beginning of the optimisation and decreased linearly to 0.4 at the maximum number of iterations [103]. The second approach is based on Clerc's constriction factor proposed in [104], where ω is not changed over time, rather it has the fixed value of 0.729, and the acceleration parameters φ_1 and φ_2 are both set to 1.49445.

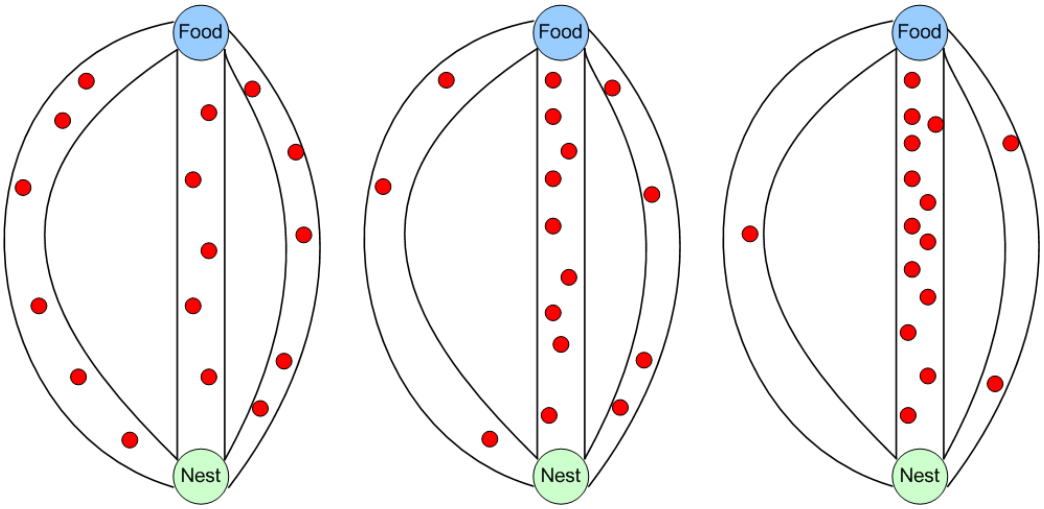
The "standard" parameter settings proposed in [104] and [103] have provided good results regarding different set of problems so that they can be used as a *guideline* in determining the best parameter configuration for the particular problem [105].

There are different improvements and refinements of the standard PSO approach [106, 107, 108, 99], which are successfully used in different domains such as machine learning [109, 110], function optimisation [111, 112, 113], chemical engineering [114] and cluster analysis [115].

Ant Colony Optimisation

Ant Colony Optimisation (ACO) has been developed by Dorigo et. al in 1996 to solve combinatorial optimisation problems, which can be reduced to the problem of finding the shortest path in a graph [30, 116]. ACO is inspired by the foraging behaviour of real ants. A real ant leaves a trail of pheromones on its path while searching for food. After finding the food, the ant follows

the pheromone trail it has laid to get back to the nest. Thus, it again lays pheromones on the same path and increases the pheromone density on it. Principally, ants are attracted by pheromones and inclined to follow the path with a high pheromone density. The pheromone density on a short path between the nest (N) and the food (F) increases more rapidly than the pheromone density on a longer path. This makes the short path more attractive to follow than the longer ones so that more and more ants follow the shorter path, while the pheromones on the longer paths vaporise after some time. Eventually, long paths disappear and only the shortest path remains (see Fig. 3.2).



(a) The ants explore the shortest path between the nest and the food.

(b) The pheromone density on the shortest path increases rapidly.

(c) Pheromones on the long paths vaporise and ants follow the shortest path.

Figure 3.2: The foraging behaviour of ants

ACO uses artificial ants to solve optimisation problems in technical systems based on the idea of laying down artificial pheromones on the path between two nodes i and j in the given graph. ACO uses a special form of indirect communication using the environment, which is called *stigmergy* [117]. Thus, ACO is also called a stigmergy-based optimisation algorithm. In most cases, the ACO algorithm utilises in addition to the positive feedback process using stigmergy also a heuristic that allows the constructive definitions of solutions [116]. The calculation of the corresponding heuristic depends on the partic-

ular problem to solve. For example, in the well-known Traveling Salesman Problem (TSP) [42] the Euclidean distance between the nodes (cities) i and j is used to implement the heuristic, which always prefers small distances over the large ones. This (greedy) heuristic is combined with the stigmergy-based approach using pheromones to calculate the probability of taking a particular path between the nodes i and j as follows 3.3:

$$p_{ij}(t) = \frac{[\tau_{ij}(t)]^{\alpha} \cdot [\eta_{ij}]^{\beta}}{\sum_{k \in allowed} [\tau_{ik}(t)]^{\alpha} \cdot [\eta_{ik}]^{\beta}} \tag{3.3}$$

In Equation 3.3, *allowed* denotes the set of all nodes which the ant can directly reach from the node i. α and β are parameters to determine the importance of the pheromone trail ($\tau_{ij}(t)$) and the heuristic (η_{ij}), respectively. As mentioned previously the pheromones are not permanent, rather they vaporise so that the probability of taking the path between the nodes i and j changes over time as shown in Equation 3.3.

ACO has been applied to different domains such as combinatorial optimisation [118, 119, 120], networking and telecommunication [121, 122, 123, 124] and data mining [125].

3.4 Summary

In this chapter, we have provided a classification of existing optimisation algorithms and presented the algorithms in more detail, which we use in our investigations. Although the presented algorithms can solve problems in various domains effectively, they have a weakness in the exploration and exploitation scheme they utilise. Here, we recognise two common patterns regarding the search behaviour of the presented algorithms. According to the first pattern, an algorithm is inclined to explore the fitness landscape at the beginning of the optimisation, and change its behaviour into exploitation step by step towards the end of the optimisation. The algorithms SA (with the decreasing temperature), ACO (with the increasing pheromone density) and PSO (with the linearly decreased inertia weight) use this pattern. The second pattern

constitutes a simultaneous exploration and exploitation behaviour in the optimisation *without* any distinction between the exploring and exploiting individuals. The algorithms GA, DE and EP use this pattern, where each individual exploit and explore at the same time according to predefined probabilities (i.e., crossover and mutation probabilities).

In OC, we deal with dynamic and self-referential fitness landscapes, where the form of the fitness landscape changes as a function of the agent behaviour requiring an adequate balance between the exploration and exploitation over the whole optimisation process. Thus, the first pattern, where the algorithm changes its behaviour from more exploration to more exploitation would fail, since the corresponding algorithm loses the capability to track the moving optimum and is not able to adapt to changes in the fitness landscape after a certain time has elapsed. In this context, the second pattern is more promising, since the algorithms using this pattern do not stop exploring the fitness landscape. However, this pattern has the weakness that all individuals explore and exploit at the same time. In this context, the exploration allows to find new higher peaks in the fitness landscape, while the exploitation of existing information results in the acceptance of solutions that are solely in the neighbourhood of the current peak. In this case, the greedy exploitation works against the creative exploration and increases the time, which is required to track the moving optimum and react on changes in the fitness landscape properly. This suggests a strict distinction between the exploring and exploiting individuals to increase the convergence rate, and reduce the time required to find higher peaks after the form of the fitness landscape has changed. Thus, there is a need of a new and effective optimisation scheme, where previously found good solutions are kept while other parts of the fitness landscape are further explored, simultaneously. This requires an intelligent distinction between exploring and exploiting individuals, which is not the case in the algorithms presented in this chapter. In the next chapter, we present the Role-based Imitation algorithm (RBI) that satisfies this requirement utilising a dynamic role assignment strategy to determine the exploring and exploiting individuals to effectively explore the fitness landscape and to find the optimum.

Chapter 4

The Role-based Imitation Algorithm

"The imagination imitates. It is the critical spirit that creates."
Oscar Wilde

One of the major issues in optimisation is the exploration/exploitation dilemma [126] with respect to determining an adequate balance between exploring and exploiting individuals to effectively find the optimum in a particular fitness landscape. In this context, the Role-based Imitation Algorithm (RBI) proposes a strict distinction between exploring and exploiting individuals (agents) providing an effective optimisation scheme, where previously found good solutions are kept while other parts of the fitness landscape are further explored simultaneously. In the following, we present RBI in detail.

4.1 A Role-based Approach to the Exploration/Exploitation Dilemma

The main idea behind RBI is to assign individuals different roles during different optimisation steps in order to guarantee (1) a sufficient exploration of

the fitness landscape over the whole optimisation process and (2) the convergence of individuals to good (possibly optimal) solutions in a short amount of time. At any point in time, the assignment of a role ("explorer" or "exploiter") is adapted with respect to (1) the current degree of convergence of a (sub-)population and (2) the relative quality of the agent's solution (see Fig. 4.1).

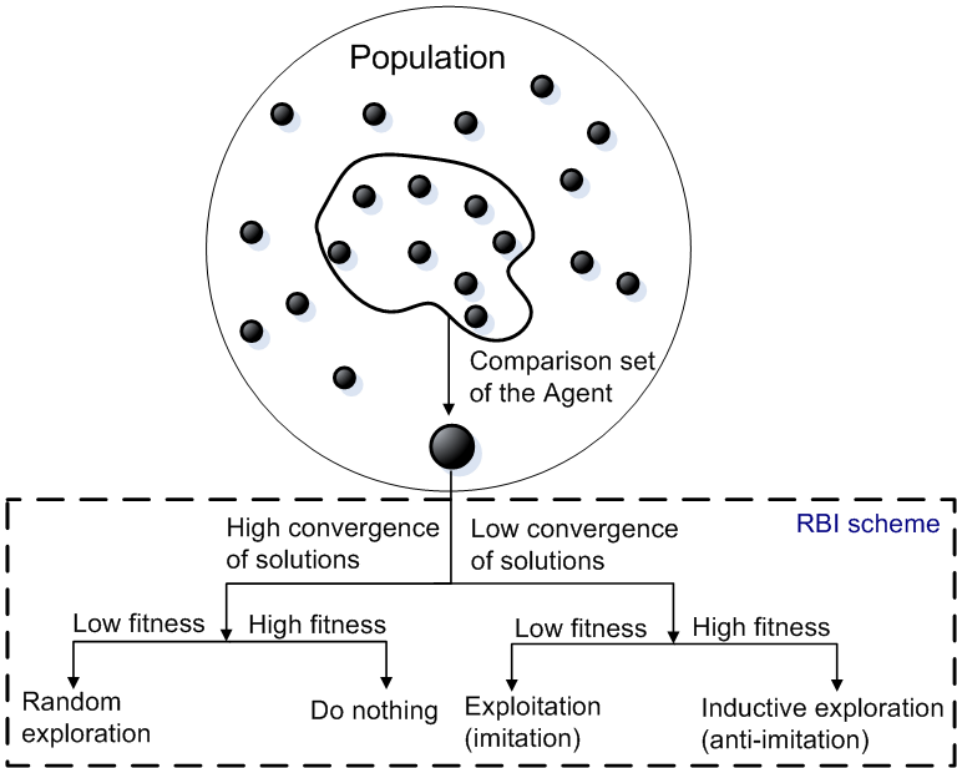


Figure 4.1: The RBI scheme defining different roles for the agents.

Fig. 4.1 shows the generic role assignment scheme, which can be implemented in discrete and continuous search spaces using different operators (e.g., arithmetic or genetic). Thus, we give first an abstract definition of the roles presented in Fig. 4.1 without explaining the detailed implementation of them in order to show the generic idea. The concrete implementation of these roles for continuous and discrete search spaces is presented in Sec. 4.2 and in Sec. 4.3, respectively.

At each optimisation step, an agent determines a random *comparison set*

of agents from the population to gather information about their solutions and the corresponding fitness values. In this context, each RBI-agent has access to the whole population and determine freely, which agent belongs to its comparison set (i.e., there is no limitation in the selection of agents for the comparison set). There are two important aspects we have to consider regarding the size of the comparison set. Firstly, the comparison set should not be too large so that the computational costs remain small and the algorithm produces good results in acceptable time. Secondly, the comparison set should not be too small so that it is not representative for the population and does not reflect e.g. the convergence state of the individuals in the population. Unfortunately, the adequate number of agents in the comparison set differs between different problem settings. Thus, a corresponding parameter study is required to determine the correct size of the comparison set for each particular problem.

After having determined the comparison set, the agent examines these solutions and distinguishes between two cases: (1) High convergence of solutions, and (2) low convergence of solutions (see Fig. 4.1).

Case 1: High convergence of solutions
In this case, the agents within the comparison set have similar solutions for the given problem, and the agent chooses one of the two possible actions *Do nothing* or *Random exploration* according to the relative quality of its solution. The idea is to use the less successful agents in the comparison set to explore the fitness landscape, while preserving the already found good solutions using the more successful ones.

1. Do nothing
   The agent does not change its solution if its fitness value is larger than the average fitness value of the agents in its comparison set maintaining a good (possibly optimal) solution.

2. Random exploration
   The agent takes on the role of an explorer, if its fitness value is less than the average fitness value of the agents in its comparison set. This

type of exploration is necessary, if we optimise a problem with e.g., a complex multimodal fitness landscape, which is mostly the case in real-world problems. In that case, the agents may have converged to a specific area in the fitness landscape, which possibly contains only (one or more) local optima. In order to avoid these local optima, the explorer agents search randomly for higher peaks (i.e., better solutions) in the fitness landscape, while the successful agents from the comparison set do not take any actions and preserve the already found good solutions.

Case 2: Low convergence of solutions

In this case, the solutions of the agents in the comparison set have not converged, and the agent takes on the role of an explorer or an exploiter according to its fitness value (see "inductive exploration" and "exploitation" in Fig. 4.1). In order to implement these two roles, each agent A_i divides other agents in its comparison set into two groups. These are (1) the agents that have a higher fitness value than A_i (the set φ), and (2) the agents that have a lower fitness value than A_i (the set λ). In this context, A_i uses the agents in φ to implement the role of the exploiter (imitator), and the agents in λ to implement the role of the explorer (anti-imitator). But before A_i can assume one of these roles, it must first determine, which role it should take on. For this purpose, A_i uses the more successful agents from the set φ, and checks whether it is also successful or not. A_i is *successful*, if its fitness value is *close enough* to the fitness values of agents that have a higher fitness value than itself, i.e., if the following condition holds:

$$fitness(A_i) > meanF(\varphi) - stdDevF(\varphi) * \beta \tag{4.1}$$

where $meanF()$ and $stdDevF()$ are functions to calculate the arithmetic mean and the standard deviation of fitness values of agents from the set φ, respectively. The control parameter β is used to determine the closeness between the fitness value of A_i and meanF (φ). According to the condition above, A_i takes on one of the two possible roles: (1) Inductive exploration and (2) exploitation.

1. Inductive exploration (anti-imitation)
   A_i is successful if condition 4.1 holds. In this case, it takes on the role of an explorer and changes its solution "away from" those agents that have a lower fitness value than itself using the set λ. The underlying assumption is that the solution of agents from the set λ are not optimal, and a "repulsion" from these solutions **might** lead to a better fitness value. Since it is not guaranteed that this kind of repulsion always provides a better fitness value, we call this step an inductive **exploration**.

2. Exploitation (imitation)
   A_i is not successful if condition 4.1 does not hold. In this case, it takes on the role of an exploiter and imitates the successful agents in its comparison set by decreasing the distance between its own solution and the solutions represented by more successful agents in φ.

The imitation and anti-imitation roles create a particular search behaviour, where successful agents "pull" less successful ones step by step towards the optimum in the corresponding fitness landscape. Overall, the optimisation scheme presented above facilitates both a coarse-grained and a fine-grained search for the optimum using the random and the inductive exploration steps, respectively. Both of them are executed over the whole optimisation process according to the current degree of convergence of the solutions in the comparison set. The other actions (roles), "Do nothing" and "imitation", are used to preserve previously found good solutions, and at the same time to exploit them in order guarantee the convergence of the whole population towards the optimum in the given fitness landscape.

In the following, we describe the implementation of the roles presented in Fig. 4.1 for continuous and discrete search spaces in more detail.

4.2 RBI for Continuous Search Spaces

In an n-dimensional search space, each RBI-agent is a solution candidate associated with an n-dimensional parameter vector consisting of real numbers. Each element i of this n-dimensional vector should be optimised such that the

whole vector consists of the "correct" values for each dimension to provide the optimal solution. The RBI-approach for the optimisation in continuous search spaces is shown in Fig. 4.2.

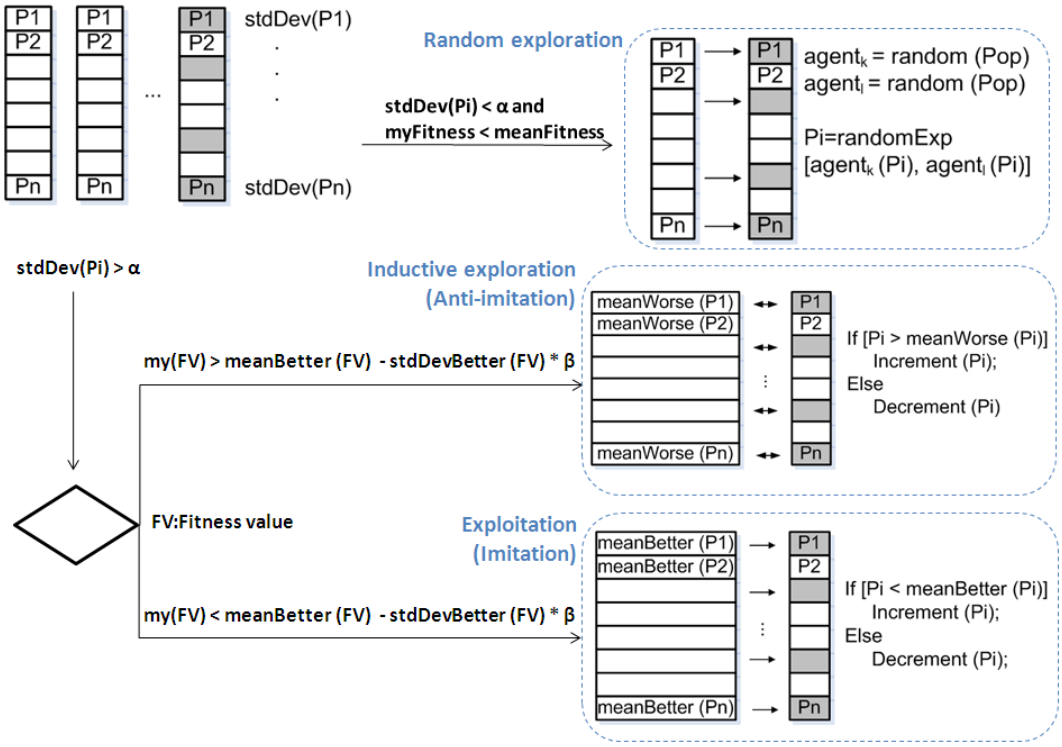


Figure 4.2: The Role-based Imitation algorithm for continuous search spaces

The solution vector of each agent is initialised randomly at the beginning of the optimisation. At each optimisation step, an RBI-agent determines (1) a random subset of dimensions to be optimised (the dimensions shown in grey in Fig. 4.2) and (2) a random comparison set of agents to gather information about their solutions and the corresponding fitness values. Afterwards, the agent calculates the standard deviation of parameter values for each considered dimension based on its comparison set, and determines whether these parameter values in particular dimensions are sufficiently converged or not. A predefined threshold α is used to determine the convergence. Generally, there is no specific rule about how to determine α. Thus, it should be adapted to the particular problem requiring a parameter study. However, in global optimisation algorithms it is usual to determine parameters such as α using the size of

the corresponding search space prior to the optimisation [6][1]. The agent takes on one of the four roles shown in Fig. 4.1 according to its fitness value and to the convergence of solutions in the considered dimension i. In the following, we present these roles in more detail.

1 - Do nothing

In this case, the agents within the comparison set have similar parameter values for the considered dimension i. Here, the agent does not change its parameter value in this dimension, and maintains a good (possibly optimal) solution if its fitness value is larger than the average fitness value of its comparison set.

2 - Random exploration

The agent takes on the role of an explorer if the agents within the comparison set have similar parameter values for the given dimension (i.e., $stdDev(P_i) < \alpha$)) and the fitness value of the agent is less than the average fitness value of its comparison set. Since the parameter values in the considered dimension are converged, there is no meaningful information available according to which the agent can change its parameter value. Thus, it chooses **randomly** two additional agents (e.g., $agent_k$ and $agent_l$) from the population that were originally **not** in its comparison set, and changes its parameter value in the i^{th} dimension using Procedure 6.

Procedure 6 The random exploration procedure

1: $probability = U[0, 1)$;
2: $x = Pi_k * (1 - probability) + Pi_l * probability$;
3: $choice = random(true, false)$
4: **if** $choice == true$ **then**
5: $\quad Pi = x + |x - max(Pi_k, Pi_l)| * U[0, 2)$
6: **else**
7: $\quad Pi = x - |x - min(Pi_k, Pi_l)| * U[0, 2)$
8: **end if**

According to Procedure 6, the agent calculates a random value (variable x) between Pi_k and Pi_l, which are the parameter values of the randomly selected agents $agent_k$ and $agent_l$ in the i^{th} dimension, respectively (line 2).

[1]For example, if the search space of a particular dimension is defined between -10 and 10, and α is set to $1/100^{th}$ of this search space, then the value of α is 0.2.

Afterwards, the agent determines where to explore using the variable *choice*, which is either $true$ or $false$ with a probability of 50% (line 3). If the variable *choice* is $true$, the agent explores around $max(Pi_k, Pi_l)$, which gives the larger parameter value of Pi_k and Pi_l. Here, the agent calculates an offset using the difference between x and $max(Pi_k, Pi_l)$ and multiplies it with a uniformly distributed random number between 0 and 2 (line 5). The parameter value of the current agent in the i^{th} dimension is determined by simply adding this offset to the variable x (line 5). Line 7 implements the exploration around $min(Pi_k, Pi_l)$, correspondingly.

3 - Inductive exploration (anti-imitation)

In this case, the parameter values of agents in the comparison set have not converged and the agent (A_1) is successful according to the condition given in Eq. 4.1. Here, A_1 takes on the role of an explorer and changes the value of its parameter in the i^{th} dimension "away from" those agents that have a lower fitness than itself using Procedure 7:

Procedure 7 The inductive exploration procedure

1: $offset = |P_i - meanP_i(\lambda)| * U[0, 2)$
2: **if** $P_i > meanP_i(\lambda)$ **then**
3: $\quad P_i = P_i + offset$
4: **else**
5: $\quad P_i = P_i - offset$
6: **end if**

where λ is the set of agents from the comparison set that have a lower fitness value than A_1, $meanP_i(\lambda)$ is the average parameter value of agents from the set λ in the i^{th} dimension and $U[0, 2)$ is a uniformly distributed random number generated between 0 and 2.

4 - Exploitation (imitation)

If the agent A_1 is not successful according to the condition given in Eq. 4.1, it takes the role of an exploiter and imitates the successful agents in its comparison set using Procedure 8.

In Procedure 8, φ is the set of agents from the comparison set that have a larger fitness value than A_1, $meanP_i(\varphi)$ is the average parameter value of agents from the set φ in the i^{th} dimension, and $U[0, 2)$ is a uniformly dis-

Procedure 8 The exploitation procedure

1: $offset = |P_i - meanP_i(\varphi)| * U[0, 2)$
2: **if** $P_i > meanP_i(\varphi)$ **then**
3: $\quad P_i = P_i - offset$
4: **else**
5: $\quad P_i = P_i + offset$
6: **end if**

tributed random number generated between 0 and 2.

Time complexity of RBI for continuous search spaces

We determine the time complexity of RBI for continuous search spaces based on the number of agents n in the comparison set and the number of dimensions m that the agent optimises at the corresponding iteration. In this context, we consider the longest execution path to determine the time complexity, where an agent must execute the following 3 steps:

1. The agent calculates the mean parameter value for each considered dimension based on its comparison set. This step has a time complexity of $O(nm)$.

2. The agent determines the standard deviation of parameter values for each considered dimension based on its comparison set. This step has the same time complexity as the step 1, which is $O(nm)$.

3. The agent calculates a new parameter value for each dimension using the Proc. 6, the Proc. 7 or the Proc. 8. The time complexity of this step is $O(m)$.

The time complexity resulting from the execution of the steps presented above is $O(2nm+m)$. Since the calculation of the average parameter values and standard deviations is the dominant time factor in RBI, we can also express the overall time complexity as $O(nm)$.

In this section, we have presented the concrete implementation of the roles defined in the RBI-scheme presented in Fig. 4.1 for the problems with search

spaces defined over continuous variables. We present in the next section 4.3 the application of the RBI-scheme to discrete optimisation problems.

4.3 RBI for Discrete Search Spaces

According to the RBI-scheme presented in Fig. 4.1, an agent determines the convergence of solutions presented by the agents in its comparison set to decide which role to take on. For search spaces defined over continuous variables, we have used the standard deviation of (partial) solutions to determine whether these solutions have converged or not. In discrete optimisation problems, we need a different distance metric for the same purpose, since the standard deviation may not be applicable to the particular type of discrete solutions such as those we have in the Traveling Salesman Problem (TSP) [42]. In RBI, we use the *Hamming distance* to determine the convergence of solutions in discrete search spaces. The Hamming distance measures the similarity between two strings of the same length by calculating the number of positions at which the corresponding symbols in the compared strings are different (see the example in Fig. 4.3).

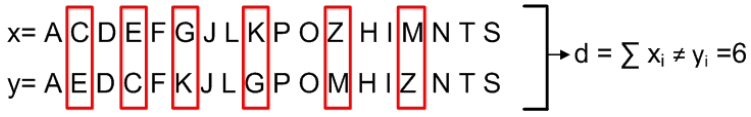


Figure 4.3: The Hamming distance between x and y is 6.

Both strings x and y in Fig. 4.3 can be considered as two different solutions for a problem that has a discrete search space. Thus, the similarity between the solutions x and y increases with the decreasing Hamming distance so that the Hamming distance is zero (and the similarity 100%) if x and y are identical. Deviations between the solutions x and y, on the other hand, lead to an increase of the Hamming distance (i.e., similarity between x and y decreases).

Another important aspect in optimisation with RBI in discrete search spaces is the implementation of exploration and exploitation procedures. If we consider the example given in Fig. 4.3 again, we notice immediately that we cannot use arithmetic operators to implement the roles defined in the RBI-

scheme presented in Fig. 4.1. This fact becomes more apparent, if we deal with combinatorial optimisation problems (e.g, TSP) rather than with the optimisation of e.g., real-valued continuous functions [23]. Thus, we need different operators to implement the exploration and exploitation procedures of RBI for the optimisation in discrete search spaces. For this purpose, we use the *genetic* operators (i.e., crossover and mutation) instead of arithmetic operators to realise the RBI-scheme. The RBI approach for the optimisation in discrete search spaces is shown in Fig. 4.4.

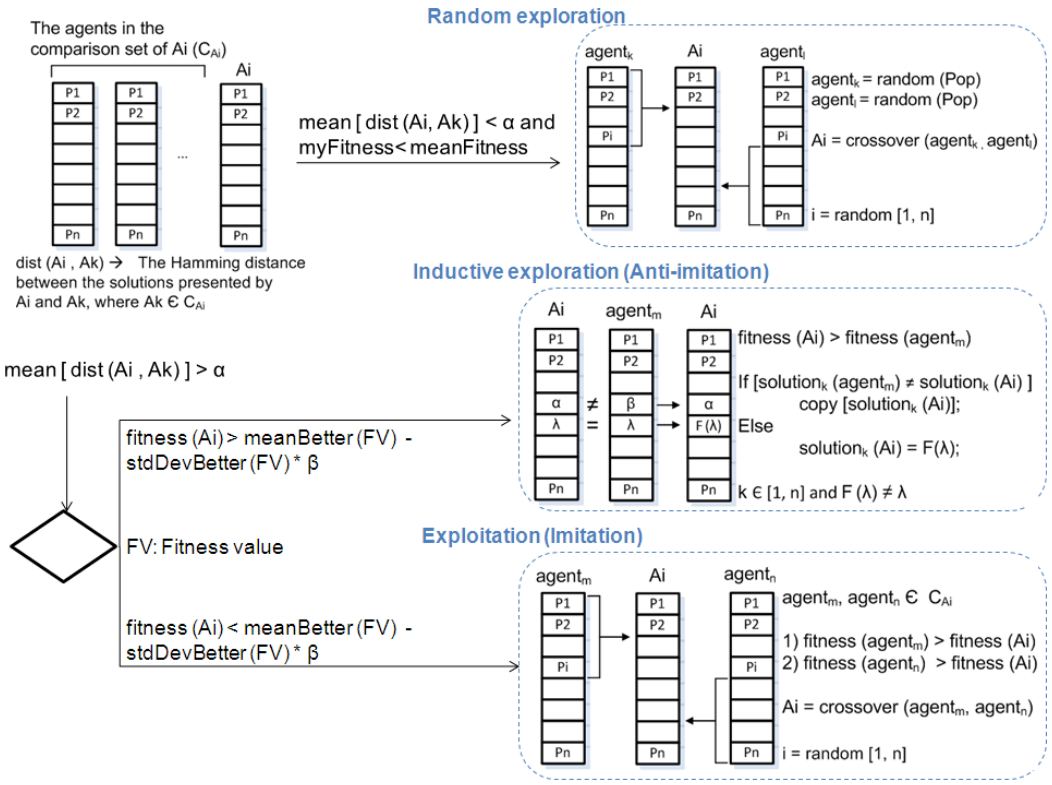


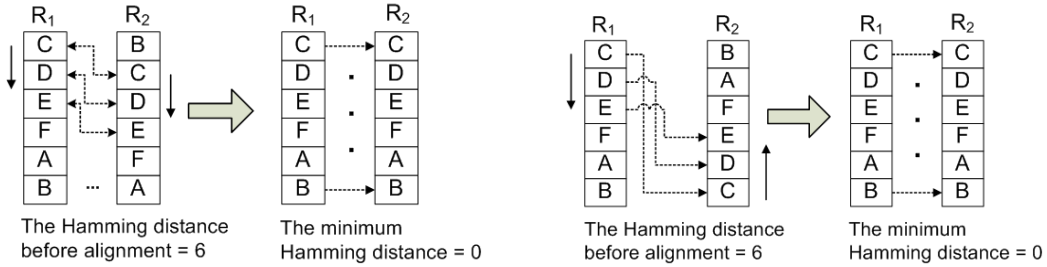
Figure 4.4: The Role-based Imitation algorithm for discrete search spaces.

Fig. 4.4 defines the main characteristics of the role assignment strategy that is used for the optimisation in discrete search spaces. However, it is not possible to provide a concrete implementation of the proposed roles, which is applicable to all kinds of discrete optimisation problems. The reason is that problem representations in discrete search spaces may significantly differ according to the particular problem to solve. Thus, different problems may require different

mutation and crossover operators to create new offspring solutions. In order to understand this matter more clearly , we consider the well-known *Boolean Satisfiability Problem* (SAT) [18] and the Traveling Salesman Problem (TSP) [42]. The goal in SAT is to find the correct *binary* solution that provides $TRUE$ according to a given boolean formula, while the goal in TSP is to find a correct *permutation* of cities, which provides the shortest route in the given topology. In SAT and TSP, we cannot use the same crossover operator to create offspring. In this context, it is possible to use the standard single-point (SPX) or multi-point (MPX) crossover operator to create offspring for the SAT problem, while TSP requires crossover operators for permutation-based representations such as Partially Match Crossover (PMX) [127] or Cycle Crossover (CX) [128] operators. Furthermore, we can use the bit-flip operator to mutate specific genes of a solution for SAT, which is not applicable to a typical solution for TSP. Since it is not possible to provide a generic concrete implementation of the roles presented in Fig. 4.4 for all discrete optimisation problems, we concentrate on TSP, which is used for benchmarking purposes to investigate RBI in this thesis.

As mentioned previously, we calculate the Hamming distance between two solutions to determine the similarity between them. Since we try to solve TSP using RBI, we consider the 1-dimensional representations (permutations) of *cities* in two given *routes* (solutions) R_1 and R_2. In order to determine the similarity between R_1 and R_2, we have to consider the alignment of the cities in these routes. For example, consider the case, where R_2 is created by rotation of the cities in R_1 by one position to the left. In such a case, the Hamming distance between R_1 and R_2 would be definitely not zero, although they present the same solution. Thus, we have to find a way to calculate the *minimum* Hamming distance between the routes R_1 and R_2 in order to determine the actual similarity between them. In this context, we consider two different cases presented in Fig. 4.5.

Fig. 4.5 shows the cases, where (1) the cities in R_2 are only "rotated" so that the minimum Hamming distance can be calculated by finding the best alignment between R_1 and R_2 (see Fig. 4.5(a)) and (2) the cities in R_2 are arranged in the reversed order of the cities in R_1 (see Fig. 4.5(b)). In the



(a) Calculate the minimum Hamming distance by finding the best alignment between R_1 and R_2

(b) Calculate the Hamming distance between R_1 and R_2 by considering the reversed order of cities in R_1

Figure 4.5: The routes R_1 and R_2 have to be compared both in straight and reversed order to calculate the minimum Hamming distance between them.

second case, we have to compare the cities in R_1 and R_2 by "moving" into opposite directions to determine the minimum Hamming distance. According to both of these aspects, we determine the similarity of a solution presented by an agent A_i to solutions presented by the agents in its comparison set using Procedure 9.

According to Procedure 9, the agent A_i calculates for each agent in its comparison set (line 5) the Hamming distance between its route R_i and the route of the corresponding agent R_j (line 6). The variable $minDistance$ holds the minimum Hamming distance between R_i and R_j, and is initialised with $INTEGER.MAX$, which defines the maximum integer number according to the corresponding programming language (line 7). After that, Procedure 9 iterates through the cities in R_i and R_j, and determines the values for the variables $distance$ (lines 12 - 14) and $reversedDistance$ (lines 15 - 17), which are used to determine the Hamming distance between R_i and R_j in straight and reversed orders, respectively[2] (see Fig. 4.3). The value of the variable $minDistance$ is changed using the minimum of the variables $distance$ and $reversedDistance$ (line 20). Afterwards, the cities in R_i are rotated one position to the left so that the first city in R_i is placed at the last position in the list of cities after the rotation (line 22). Thus, in the next iteration we

[2]The cities in R_i and R_j are indexed starting from 0 so that the index of the last city in each route is $numberofCities - 1$. Thus, the method $getCity$ returns the same city for the indices 0 and $numberofCities$.

Procedure 9 The calculation of Hamming distance for TSP

```
1: Define C_Ai                                   ▷ The comparison set of A_i.
2: int meanHamming = 0
3: R_i = getRoute(A_i)                           ▷ The route (solution) of A_i.
4:
5: for all (A_j ∈ C_Ai) do
6:     R_j = getRoute(A_j)                       ▷ The route (solution) of A_j.
7:     int minDistance = INTEGER.MAX
8:     for (n = 0 to numberofCities − 1) do
9:         for (m = 0 to numberofCities − 1) do
10:            int distance = 0
11:            int reversedDistance = 0;
12:            if (getCity(R_i, m)! = getCity(R_j, m)) then
13:                ++distance
14:            end if
15:            if (getCity(R_i, m)! = getCity(R_j, numberofCities − m)) then
16:                ++reversedDistance
17:            end if
18:        end for
19:        if (min[distance, reversedDistance] < minDistance) then
20:            minDistance = min[distance, reversedDistance]
21:        end if
22:        rotate(R_i)          ▷ Rotates the cities in R_i one position to the left.
23:
24:    end for
25:    meanHamming+ = minDistance
26: end for
27: meanHamming/ = numberofAgents(C_Ai)
```

calculate *distance* and *reversedDistance* beginning from the next city in R_i. Eventually, the value of the variable *meanHamming* is calculated using the number of agents in C_{Ai} (line 26).

The variable *meanHamming* holds the average Hamming distance between the route of A_i and the routes of agents in its comparison set. Here, we use again the predefined parameter α in order to determine the convergence of solutions (see Fig. 4.4). If the value of the variable *meanHamming* is less than α, we conclude that the solution represented by A_i and the solutions represented by the agents in its comparison set have sufficiently converged so

that A_i either takes no further actions (the role "Do Nothing") or it takes on the role of a random explorer depending on its fitness value. If the value of the variable $meanHamming$ is larger than α, we conclude that there are agents in the comparison set of A_i, whose solutions significantly differ from the solution of A_i so that it can perform either the inductive exploration or the exploitation step.

In the following, we present the implementation of each role for TSP:

1 - Do nothing

In this case, the solutions represented by the agents in the comparison set of A_i have sufficiently converged, and the fitness value of A_i is *larger* than the average fitness value of its comparison set. Here, A_i takes no further actions and maintains a good (possibly optimal) solution.

2 - Random Exploration

In this case, the solutions represented by the agents in the comparison set of A_i have sufficiently converged, and the fitness value of A_i is *less* than the average fitness value of its comparison set. In this situation, there is a high correspondence between the routes represented by the agents so that A_i has no meaningful information in its comparison set to optimise its route. Hence, it selects *randomly* two additional agents from the population that were originally *not* in its comparison set to explore the fitness landscape. The random exploration is implemented using the Partially Match Crossover (PMX) [127] operator with the randomly selected agents as parents. PMX is a special type of two-point crossover operator, which inherits edges between the selected crossover points and, at the same time introduces *new edges* outside of these points. This property makes PMX very suitable for the implementation of random exploration. PMX is implemented as shown in Fig. 4.6.

The PMX-operator first selects two crossover points for the parents (step 1). The substrings selected in Parent 1 and Parent 2 are used to create a mapping between the components of these substrings (step 2). After that, two offspring, Offspring 1 and Offspring 2, are created from Parent 2 and Parent 1, respectively by simply copying the genes of parents between the selected crossover points to the offspring (step 3). At step 4, the routes represented by the offspring are completed. There, the components from Parent 1 are copied

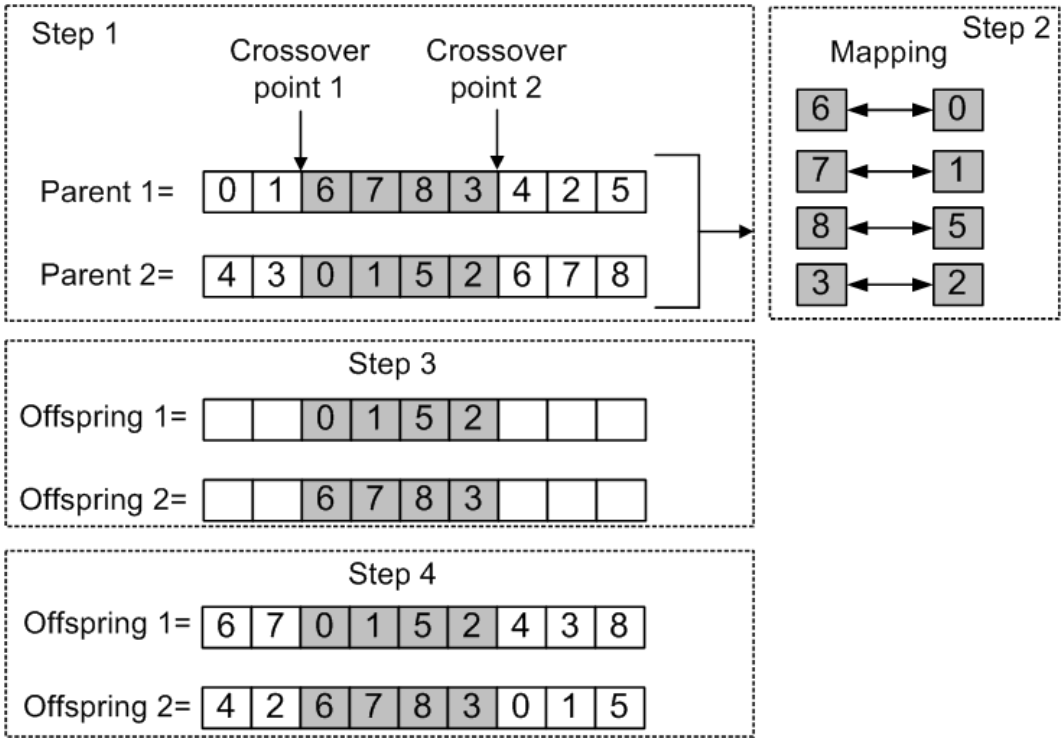


Figure 4.6: The Partially Match Crossover (PMX) operator.

to Offspring 1 (and also from Parent 2 to Offspring 2) as long as they are not part of the mapping created in step 2. Otherwise, PMX uses the mapping to decide which component to add to the offspring at the corresponding position. For example, we cannot copy 0 from parent 1 to offspring 1 directly because 0 is mapped to 6 at step 2. Thus, 6 is copied into the offspring instead 0 at the corresponding position. At the end, the PMX-operator creates two offspring, and the random explorer selects one of them at random.

3 - Inductive exploration (anti-imitation)

In this case, the solutions represented by the agents in the comparison set of A_i have *not* sufficiently converged, and A_i is successful according to the condition given in Eq. 4.1. Thus, A_i assumes the role of an explorer according to the RBI-scheme presented in Fig. 4.4. In this context, A_i selects an agent from its comparison set (e.g., A_j) randomly, which has a lower fitness value than itself, and increases the Hamming distance between its solution and the solution of A_j as shown in Fig: 4.7.

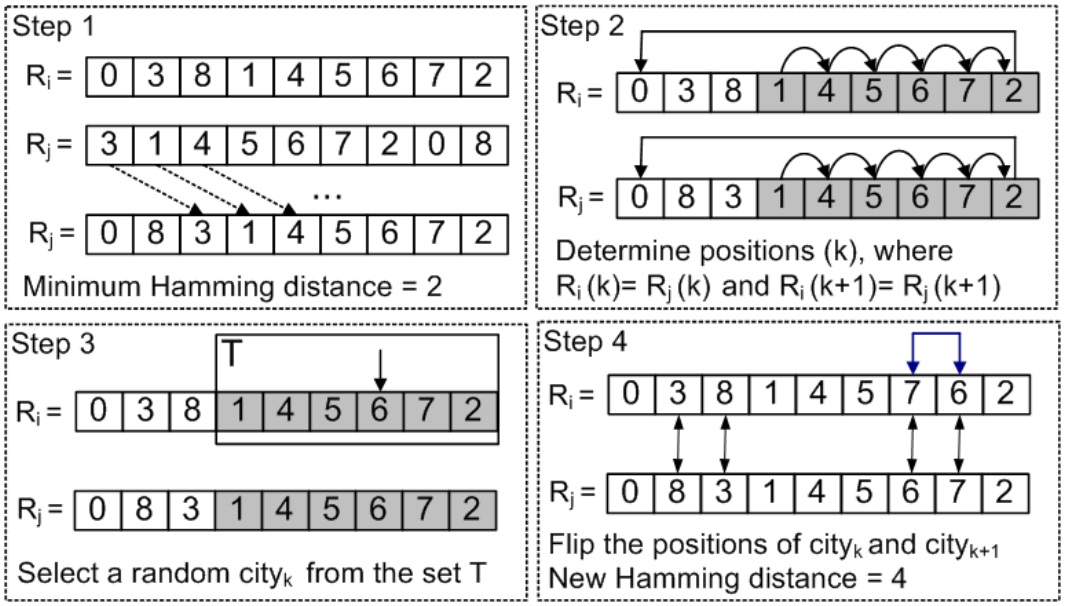


Figure 4.7: The inductive exploration (anti-imitation) for TSP

In the first step, the minimum Hamming distance between the routes R_i and R_j is determined, which are the routes represented by A_i and A_j, respectively (step 1). After that, A_i determines the positions in both routes, where $R_i(k) = R_j(k)$ and $R_i(k+1) = R_j(k+1)$ (step 2). This step identifies the same transitions between the cities of R_i and R_j, and saves them into the set T. After selecting a random city (city_k) from T at step 3, A_i flips the positions of city_k and city_{k+1}. Based on this action, A_i explores the fitness landscape by increasing the Hamming distance between R_i and R_j and *moving away* from the solution represented by A_j.

The inductive exploration resembles the well-known swap mutation operator for TSP [129]. The difference between both approaches is that the swap mutation operator involves only a single solution and flips two cities in a given route randomly without considering an additional route. The inductive exploration, on the other hand, determines the cities to flip not completely at random, rather according to an additional (worse) solution, and thus influences the direction of mutation.

4 - Exploitation (imitation)

In this case, the solutions represented by the agents in the comparison set of

A_i have *not* sufficiently converged, and A_i is not successful according to the condition given in Eq. 4.1. Here, A_i assumes the role of an exploiter, and imitates two randomly selected agents (parent1 and parent2) with a better fitness value from its comparison set using the special two-point *Nearest Neighbour Crossover* operator [130] as shown in Procedure 10.

Procedure 10 The exploitation (imitation) procedure for TSP

```
 1: Define parent1                                          ▷ The first parent
 2: Define parent2                                          ▷ The second parent
 3: Define offspring                                        ▷ The new solution
 4: int crossoverPoint1 = 0
 5: int crossoverPoint2 = 0
 6:
 7: while (crossoverPoint1 ≥ crossoverPoint2) do
 8:     crossoverPoint1 = random(numberofCities)
 9:     crossoverPoint2 = random(numberofCities)
10: end while
11: offspring = copy(parent1, crossoverPoint1, crossoverPoint2)
12: for (i = crossoverPoint2 − crossoverPoint1 to numberofCities) do
13:     currentCity = getCity(offspring, i)
14:     adjacentCities[] = getAdjacentCity(parent2, currentCity)
15:     if (¬offspring.contains(adjacentCities[1])) then
16:         offspring = add(adjacentCities[1])
17:     else if (¬offspring.contains(adjacentCities[2])) then
18:         offspring = add(adjacentCities[2])
19:     else
20:         offspring = add(nearestNeighbour(currentCity))
21:     end if
22: end for
```

As shown in Procedure 10, we first determine two crossover points (lines 7-10) in order to copy the corresponding part of *parent*1 into the offspring (line 11). After that, we complete the route of the offspring using the cities of *parent*2. Here, we consider the *last* city copied into the offspring (*currentCity* in line 13), and determine the cities in *parent*2 that are adjacent to *currentCity* (line 14). There are two adjacent cities to *currentCity* in *parent*2[3], and if one of them is not in the offspring, we add the corresponding city into the offspring

[3]Each city in a given TSP is connected to exactly two different cities.

(line 16 or line 18). Otherwise, if both adjacent cities in *parent*2 are already in the offspring, we find the nearest neighbour of *currentCity* in the given TSP, which is not in the offspring and use it to complete the route of the offspring (line 20). Thus, Procedure 10 creates an offspring, which contains as many edges as possible from both parents, and therefore is very suitable to implement the exploitation step for TSP.

Time complexity of RBI for discrete search spaces

The dominating factor in the time complexity of RBI is the calculation of the Hamming distance presented in Procedure 9. According to Procedure 9, an agent A_i calculates the average minimum Hamming distance between its own route and the routes represented by the agents in its comparison set resulting in a time complexity of $O(nm^2)$, where n is the number of agents in the comparison set of A_i and m is the number of cities in the given topology. At this point, we emphasise again that we concentrate on the optimisation of TSP in this thesis. Therefore, the implementation of the roles presented in Fig. 4.1 is adapted to TSP together with Procedure 9, which is used to determine the convergence of solutions in the comparison set. Hence, the time complexity of RBI would be different (possibly less) than $O(nm^2)$, if we optimise a different problem, e.g., SAT, where we can omit the steps required to find the best alignment between the solutions presented in Procedure 9.

4.4 Summary

In this chapter, we have presented the Role-based Imitation algorithm (RBI) for search spaces defined over continuous and discrete variables. RBI uses an intelligent role assignment strategy to determine the exploring and exploiting agents in the population in order to establish a balance between the creative exploration process and the greedy exploitation process. For both types of search spaces we have provided a detailed implementation of the roles presented in Fig. 4.1. The implementation of RBI for problems defined over continuous variables is generic, since we can express the corresponding technical problems

as n-dimensional continuous optimisation problems (e.g., as function optimisation [23]), where the goal is to find a real-valued n-dimensional vector as the complete solution. On the other hand, the problem representation is not unique in case of optimisation in discrete search spaces (e.g., the optimisation of problems such as SAT and TSP) so that the concrete implementation of the roles must be defined and adapted to the particular problem. In this context, we have investigated the well-known Traveling Salesman Problem (TSP) that is widely used for benchmarking purposes, and provided the implementation of the roles given in Fig. 4.1 for TSP.

According to the information provided in this chapter, we now can extend our evaluation strategy presented in Sec. 2.4 as shown in Fig. 4.8.

	DE	PSO	GA	SA	ACO	RBI
Applicable in						
continous search spaces	Yes	Yes	Yes	Yes	No	Yes
discrete search spaces	No	No	Yes	Yes	Yes	Yes
Continous search spaces						
Static fitness landscapes						
Solution quality in noiseless env.						
Solution quality in noisy env.						
Convergence speed						
Self-referential fitness landscapes						
Solution quality in noiseless env.						
Solution quality in noisy env.						
Convergence speed						
Discrete search spaces: TSP						
Static fitness landscapes						
Solution quality in noiseless env.						
Convergence speed						

Figure 4.8: The evaluation strategy used to compare DE, PSO, GA, SA, ACO and RBI.

OC systems may have different kinds of fitness landscapes such as static, where the solution-fitness mapping does not change over time, or dynamic and self-referential, where the form of the fitness landscape changes as a function

of the agent behaviour over time [26, 27]. Thus, an optimisation algorithm, which is designed to optimise the behaviour of an OC system should work efficiently in both types of fitness landscapes. In this context, we investigate in Chap. 5 and Chap. 6 the performance of RBI in static and self-referential fitness landscapes providing a comprehensive comparison of RBI to the state-of-the-art optimisation algorithms presented in Chap. 3.

Chapter 5

Optimisation in Static Fitness Landscapes

In this chapter, we investigate the performance of RBI in static fitness landscapes, where a given solution produces always the same fitness value regardless of time or agent behaviour. In this context, we provide a comprehensive comparison of RBI to the well-known state-of-the-art optimisation algorithms presented in Chapter 3. Here, we investigate RBI (1) in search spaces defined over continuous variables using standard benchmark problems from the domain of *function optimisation*, and (2) in search spaces defined over discrete variables using the *Traveling Salesman Problem (TSP)*. The goal in function optimisation is to find the solution that provides the minimum function value. In this context, we compare RBI to Differential Evolution (DE), Particle Swarm Optimisation (PSO), Genetic Algorithm (GA) and Simulated Annealing (SA). The corresponding experimental results are presented in Sec. 5.1. Similarly, the goal in TSP is to find the shortest route in the given topology, which presents also a minimisation problem. In this context, we compare in Sec. 5.2 RBI to Genetic Algorithm (GA), Simulated Annealing (SA), Ant Colony Optimisation (ACO), Evolutionary Programming (EP) and Annealing Genetic Algorithm (AG), and present the experimental results. Finally, in Sec. 5.3 we summarise the results presented in this chapter and discuss the advantages/disadvantages of the algorithms regarding the investigated set of

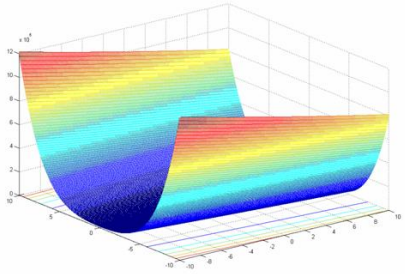
problems.

5.1 Function Optimisation with RBI

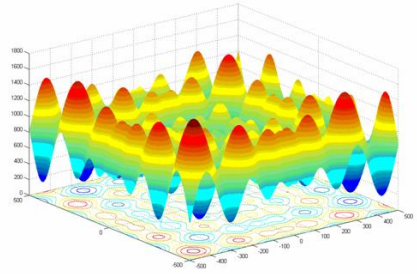
There are two main evaluation criteria that are used to compare the performances of optimisation algorithms: (1) The quality of solutions provided by the optimisation algorithm and (2) the convergence speed of the optimisation algorithm. In order to investigate RBI according to these two criteria, we have implemented 21 different benchmark functions with unimodal, multimodal and noisy fitness landscapes, where each optimisation algorithm tries to find the global **minimum**. The benchmark functions are taken from [23]. The implemented functions can be classified into two groups: (1) High-dimensional and (2) low-dimensional functions. We investigate the high-dimensional functions (see Fig. 5.2) in two different levels of complexity by implementing them (1) in 30 and (2) in 50 dimensions. The low dimensional functions, on the other hand, have only 2 or 4 dimensions (see Fig. 5.3). The rows shown in gray in Fig. 5.2 and in Fig. 5.3 contain the parameter information required to calculate the high-dimensional functions f_{12} and f_{13} and the low-dimensional functions f_{14}, f_{15}, f_{19}, f_{20} and f_{21}, respectively. Fig. 5.1 shows some of the investigated functions[1].

Regarding the high-dimensional functions from f_1 to f_{13}, the first seven functions (f_1 - f_7) have unimodal and the remaining functions (f_8 - f_{13}) have multimodal fitness landscapes. Since these functions are all high-dimensional, it is not trivial to find the minimum in the corresponding search space requiring an intelligent exploration and exploitation scheme due to the large size of the parameters to be optimised. In this context, the multimodal functions (f_8 - f_{13}) present an additional challenge to the optimisation algorithms, since the number of local minima increases with the dimensionality of the corresponding function. Thus, in this case an optimisation algorithm must not only cope with the high-dimensionality of the functions but also with the difficulties regarding the form of the corresponding fitness landscape. The function f_7, which is a quartic noisy function with a unimodal fitness landscape, is of

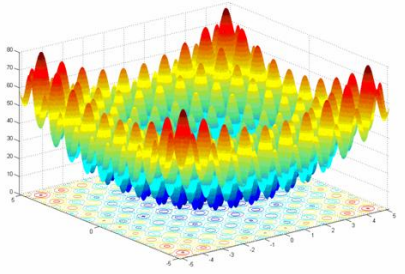
[1]The figures are taken from `http://www-optima.amp.i.kyoto-u.ac.jp/`.



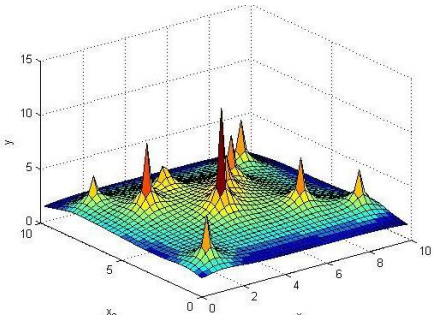
(a) F5 is the Rosenbrock function, which is unimodal and high dimensional.



(b) F8 is the Schwefel function, which is multimodal and high dimensional.



(c) F9 is the Rastrigin function, which is multimodal and high dimensional.



(d) F14 is the Shekel function, which is multimodal and 2 dimensional.

Figure 5.1: The three dimensional representations of Rosenbrock (F5), Schwefel (F8), Rastrigin (F9) and Shekel (F14) functions each with a different type of fitness landscape.

special interest in our investigation, since noise models real-world effects like disturbances (e.g., defective sensors) that are important in the context of OC. As shown in Fig. 5.2, the calculation of f_7 involves a noise component, which is implemented using a random number generator. A uniformly distributed random number between 0 and 1 is added to the fitness value of a solution (or of an agent in case of a population-based optimisation algorithm) making the prediction of the real form of the corresponding fitness landscape very hard (or impossible).

Thus, the optimisation in this kind of fitness landscapes is a great challenge due to the inconsistencies between the fitness values of solutions determined at different points in time. The fitness landscape of f_7 is not dynamic, since

High-dimensional functions	Ranges	Minimum Value
$f_1 = \sum_{i=0}^{n-1} x_i^2$	-5.12 < xi < 5.12	0
$f_2 = \sum_{i=0}^{n-1} \lvert x_i \rvert + \prod_{i=0}^{n-1} \lvert x_i \rvert$	-10 < xi < 10	0
$f_3 = \sum_{i=0}^{n-1} \left(\sum_{j=1}^{i} x_j \right)^2$	-100 < xi < 100	0
$f_4 = \max \lvert x_i \rvert, 0 = i < n$	-100 < xi < 100	0
$f_5 = \sum_{i=0}^{n-2} (100(x_{i+1} - (x_i)^2)^2 + (x_i - 1)^2)$	-30 < xi < 30	0
$f_6 = \sum_{i=0}^{n-1} \left(\lfloor x_i + \frac{1}{2} \rfloor \right)^2$	-100 < xi < 100	0
$f_7 = \left(\sum_{i=0}^{n-1} (i+1) x_i^4 \right) + rand[0,1)$	-1.28 < xi < 1.28	0
$f_8 = \sum_{i=0}^{n-1} - x_i \sin(\sqrt{\lvert x_i \rvert})$	-500 < xi < 500	-12569.481/ -20949.145
$f_9 = \sum_{i=0}^{n-1} (x_i^2 - 10\cos(2\pi x_i) + 10)$	-5.12 < xi < 5.12	0
$f_{10} = -20\exp\left(-0.2\sqrt{\frac{1}{n}\sum_{i=0}^{n-1} x_i^2}\right) - \exp\left(\frac{1}{n}\sum_{i=0}^{n-1}\cos(2\pi x_i)\right) + 20 + e$	-32 < xi < 32	4.4408 e-16
$f_{11} = \frac{1}{4000}\left(\sum_{i=0}^{n-1} x_i^2\right) - \left(\prod_{i=0}^{n-1}\cos(\frac{x_i}{\sqrt{i+1}})\right) + 1$	-600 < xi < 600	0
$f_{12} = \frac{\pi}{n}\{10(\sin(\pi y_1))^2 + \sum_{i=0}^{n-2}((y_i - 1)^2(1 + 10(\sin(\pi y_{i+1}))^2)) + (y_n - 1)^2\} + \sum_{i=0}^{n-1} u(x_i, 10, 100, 4)$	-50 < xi < 50	1.5705 e-32/ 9.4232 e-33
$f_{13} = 0.1\{10(\sin(3\pi x_1))^2 + \sum_{i=0}^{n-2}((x_i - 1)^2(1 + (\sin(3\pi x_{i+1}))^2)) + (x_n - 1)(1 + (\sin(2\pi x_n))^2)\} + \sum_{i=0}^{n-1} u(x_i, 10, 100, 4)$	-50 < xi < 50	-1.1504403
Function „u“ for f_{12} and f_{13}		
$u(x_i a, k, m) = \begin{cases} k(x_i - a)^m, x_i > a \\ 0, -a \le x_i \le a \\ k(-x_i - a)^m, x_i < -a \end{cases}$		

Figure 5.2: The high-dimensional functions. These functions are implemented in 30 and 50 dimensions. The functions f_8 and f_{12} have different minimum values in 30 and 50 dimensions.

the fitness function *without* noise ($\sum_{i=0}^{n-1}(i+1)x_i^4$) gives for a solution s always the same fitness value $f_7(s)$ regardless of time. However, we can identify the fitness landscape of f_7 *with* noise as *quasi dynamic*, since the noise component simulates a (random) change in the form of the fitness landscape. Thus, this type of fitness landscapes can be considered as a preliminary stage of real dynamic or self-referential fitness landscapes. Since OC systems are designed to work in noisy environments [24, 33], it is very important for an optimisation algorithm to cope with the existing noise in the environment and to find the optimal behaviour at any given point in time.

The functions from f_{14} to f_{21} are low-dimensional multi-modal functions, where the number of dimensions (i.e., the number of local minima in the fitness landscape) is fixed. The functions f_{14}, f_{16}, f_{17} and f_{18} have 2 dimensions, and the functions f_{15}, f_{19}, f_{20} and f_{21} have 4 dimensions. These functions are easier to optimise in comparison to functions presented in Fig. 5.2.

Based on these 21 benchmark functions, we investigate the performance of RBI in comparison to the performances obtained by DE, PSO, GA, and SA in noiseless and noisy environments in Sections 5.1.2 and 5.1.3, respectively. Furthermore, we provide experimental results for the convergence speed of RBI in Sec. 5.1.4. Before we discuss the results in more detail, we present in the next section 5.1.1 the parameter settings we use for each optimisation algorithm.

5.1.1 Parameter Settings

In this section, we present the parameter settings used for DE, PSO, GA, SA and RBI. Generally, it is not possible to define a parameter set for an optimisation algorithm, which produces always the optimal solution for each existing problem. Thus, if we want to optimise a given set of problems using a particular optimisation algorithm, we must test and evaluate different parameter settings in order to determine the optimal setting for this specific set of problems. However, there exist parameter settings for some algorithms, which are empirically proved to be adequate for the majority of problems. Hence, these "standard" settings provide a good starting point for the comparison of

Low-dimensional functions (2 - 4 dimensions)	Ranges	Minimum Value
$f_{14} = \left(\frac{1}{500} + \sum_{j=0}^{24} \left(j + 1 + \sum_{i=0}^{1} (x_i - a_{ij})^6 \right)^{-1} \right)^{-1}$	-65.54 < xi <65.54	0.9980038
$f_{15} = \sum_{i=0}^{10} \left(a_i - \frac{x_0 (b_i^2 + b_i x_1)}{b_i^2 + b_i x_2 + x_3} \right)^2$	-5 < xi < 5	0.0003074
$f_{16} = 4x_0^2 - 2.1x_0^4 + \frac{1}{3}x_0^6 + x_0 x_1 - 4x_1^2 + 4x_1^4$	-5 < xi < 5	-1.0316
$f_{17} = \left(x_1 - \frac{5.1}{4\pi^2} x_0^2 + \frac{5}{\pi} x_0 - 6 \right)^2 + 10\left(1 - \frac{1}{8\pi}\right)\cos(x_0) + 10$	-5 < xi < 15	0,3978873
$f_{18} = \{1 + (x_0 + x_1 + 1)^2 (19 - 14x_0 + 3x_0^2 - 14x_1 + 6x_0 x_1 + 3x_1^2)\}\{30 + (2x_0 - 3x_1)^2 (18 - 32x_0 + 12x_0^2 + 48x_1 - 36x_0 x_1 + 27x_1^2)\}$	-2 < xi < 2	3.0
$f_{19} = -\sum_{i=0}^{4} \left((x - a_i)^T (x - a_i) + c_i \right)^{-1}$	0 < xi < 10	-10.532
$f_{20} = -\sum_{i=0}^{6} \left((x - a_i)^T (x - a_i) + c_i \right)^{-1}$	0 < xi < 10	-10.40294
$f_{21} = -\sum_{i=0}^{9} \left((x - a_i)^T (x - a_i) + c_i \right)^{-1}$	0 < xi < 10	-10.5364
f_{14} : $a = \begin{bmatrix} -32 & -16 & 0 & 16 & 32 & -32 & -16 & 0 & 16 & 32 & -32 & -16 & 0 & 16 & 32 & -32 & -16 & 0 & 16 & 32 & -32 & -16 & 0 & 16 & 32 \\ -32 & -32 & -32 & -32 & -32 & -16 & -16 & -16 & -16 & -16 & 0 & 0 & 0 & 0 & 0 & 16 & 16 & 16 & 16 & 16 & 32 & 32 & 32 & 32 & 32 \end{bmatrix}$		
f_{15} : $a = [0.1957\ 0.1947\ 0.1735\ 0.1600\ 0.0844\ 0.0627\ 0.0456\ 0.0342\ 0.0323\ 0.0235\ 0.0246]$ $b = \left[\frac{1}{0.25}, \frac{1}{0.5}, 1, \frac{1}{2}, \frac{1}{4}, \frac{1}{6}, \frac{1}{8}, \frac{1}{10}, \frac{1}{12}, \frac{1}{14}, \frac{1}{16} \right]$		
$f_{19\text{-}21}$: $a = \begin{bmatrix} 4.0 & 4.0 & 4.0 & 4.0 \\ 1.0 & 1.0 & 1.0 & 1.0 \\ 8.0 & 8.0 & 8.0 & 8.0 \\ 6.0 & 6.0 & 6.0 & 6.0 \\ 3.0 & 7.0 & 3.0 & 7.0 \\ 2.0 & 9.0 & 2.0 & 9.0 \\ 5.0 & 5.0 & 3.0 & 3.0 \\ 8.0 & 1.0 & 8.0 & 1.0 \\ 6.0 & 2.0 & 6.0 & 2.0 \\ 7.0 & 3.6 & 7.0 & 3.6 \end{bmatrix}$ $c = [0.1,\ 0.2,\ 0.2,\ 0.4,\ 0.4,\ 0.6,\ 0.3,\ 0.7,\ 0.5,\ 0.5]$		

Figure 5.3: The low-dimensional functions. The functions f_{14}, f_{16}, f_{17} and f_{18} are 2-dimensional and the functions f_{15}, f_{19}, f_{20} and f_{21} are 4-dimensional.

different parameter settings for a given set of benchmark problems. Therefore, we adopted the existing good parameter settings for the algorithms from the literature where possible.

The Parameter Setting for DE

We use the DE/Rand/1/exp scheme for DE, which is the most universally applicable DE-scheme that is also used in [23], where the population size is set to 100, the crossover constant (CR) is set to 0.9 and the scaling factor (F) is set to 0.5.

The Parameter Setting for PSO

A "standard" parameter setting for PSO is proposed in [103], where the population size is 30 and the inertia weight (ω) is 0.729, while the particle increment (φ_1) and the neighbourhood increment (φ_2) are both set to 1.49445. We have compared this parameter setting to the one proposed in [105], where ω is also 0.729, but φ_1 is set to 2.0412 and φ_2 is set to 0.9477. We use the parameter setting proposed in [105] with the population size of 100 particles and a neighbourhood size of 25 particles, since it produces the best results for PSO regarding the benchmark functions presented in Fig. 5.2 and in Fig. 5.3.

The Parameter Setting for GA

We use here the parameter setting proposed in [23], where the population size is set to 100 and the probabilities of mutation (p_m) and crossover (p_c) are set to 0.9 and 0.7, respectively. The mutation is implemented using the Cauchy mutation operator with an annealing scheme, while the crossover is implemented using the arithmetic crossover operator. Furthermore, a tournament selection with a tournament size of two individuals is used to select individuals for crossover.

The Parameter Setting for SA

In contrast to PSO, it is hard to find a "standard" parameter setting for SA in the literature, which can be adopted and used to optimise the large number of

benchmark functions we investigate in this section. Thus, we have performed an intensive parameter study to determine the optimal parameter setting for SA according to the given set of benchmark functions and according to the maximum number of function evaluations, which is set for each algorithm to 500,000. There are three main parameters to set for SA: (1) The initial temperature, (2) the cooling rate and (3) a distribution function to define a neighbourhood of the current solution in order to select a new trial solution. In this context, the initial temperature is set to 10e+6 and the cooling rate is set to 0.99. In order to determine the best distribution function, we have tested the Gaussian, equal, and Adaptive Simulated Annealing (ASA) [131] distributions each with an annealing scheme. The best results are obtained with the ASA distribution function.

The Parameter Setting for RBI

We have also performed an intensive parameter study in order to determine the best parameter setting for RBI. Here, we set the population size to 100 and the neighbourhood size (i.e., the size of the comparison set) to 5. The parameter α, which is used to determine the convergence in a particular dimension (see Fig. 4.2), is set to $1/100^{th}$ of the search space of the corresponding dimension (e.g., in case of f_{12} with a search range from -50 to 50, α is set to 1). The parameter β, which is used to select one of either the inductive exploration (anti-imitation) or the exploitation (imitation) steps, is set to 1.5 (see Eq. 4.1).

5.1.2 RBI in Noiseless Environments

In this section, we optimise the benchmark functions presented in Fig. 5.2 and in Fig. 5.3 with DE, PSO, GA, SA and RBI using the parameter settings presented in Sec. 5.1.1. Here, the high-dimensional functions are investigated both in 30 and 50 dimensions, while the low-dimensional functions have a fixed number of dimensions that is either 2 or 4. The maximum number of function evaluations is set to 500,000 i.e., the corresponding optimisation algorithm cannot calculate the fitness value of a solution using a particular function

more than 500,000 times. 30 experiments, each initialised with a different random seed, are carried out for each algorithm and the average best fitness values are recorded. Fig. 5.4, Fig. 5.5 and Fig. 5.6 show the experimental results for the low-dimensional functions and the high-dimensional functions with 30 and 50 dimensions, respectively. (In the rest of this chapter, the best results obtained from the optimisation of functions are presented in grey. The results below 10^{-45} are reported as "0.0000000e+00").

Results for the Low-dimensional Functions

	RBI	DE	PSO	GA	SA	Optimum
F14	9.9800384 e-01	9.9800384 e-01	9.9800384 e-01	1.6892421 e+00	9.9800384 e-01	9.9800384 e-01
F15	3.0748593 e-04	4.6010056 e-04	3.0748599 e-04	3.0814498 e-04	6.1205233 e-04	3.0748593 e-04
F16	-1.0316285 e+00	-1.0316285 e+00	-1.0316285 e+00	-1.0316285 e+00	-1.0316285 e+00	-1.0316285 e+00
F17	3.9788735 e-01	3.9788735 e-01	3.9788735 e-01	3.9788735 e-01	3.9788735 e-01	3.9788735 e-01
F18	3.0 e+00	3.0 e+00	3.0 e+00	3.0 e+00	3.0 e+00	3.0 e+00
F19	-9.486763 e+00	-10.1532 e+00	-9.735774 e+00	-8.410121 e+00	-9.984785 e+00	-10.1532 e+00
F20	-10.402941 e+00	-10.402941 e+00	-10.402941 e+00	-10.402941 e+00	-10.049961 e+00	-10.402941 e+00
F21	-10.536409 e+00	-10.536409 e+00	-10.536409 e+00	-9.772439 e+00	-10.536409 e+00	-10.536409 e+00

Figure 5.4: The averaged best fitness values obtained by DE, PSO, GA, SA and RBI for the low-dimensional functions shown in Fig. 5.3. The functions f_{14}, f_{16}, f_{17} and f_{18} have 2 dimensions and the functions f_{15}, f_{19}, f_{20} and f_{21} have 4 dimensions. Best solutions are shown in grey.

The results in Fig. 5.4 show that all algorithms produce similar results for the low-dimensional functions, while RBI, DE and PSO are slightly better than GA and SA. RBI and DE find almost in all cases (except f_{19} for RBI and f_{15} for DE) the optimal solution so that both algorithms perform on the same level in optimisation of low-dimensional functions. PSO provides very similar results to RBI especially for function f_{15}. In this case, PSO finds a solution, which is very close to the solution found by RBI so that the difference between both fitness values is less than 10^{-6}. GA gets stuck in a local optimum on functions f_{14}, f_{19} and f_{21}, while it finds the same solutions as RBI, DE and PSO on functions f_{16}, f_{17}, f_{18} and f_{20}. SA is more or less on the same level as GA and cannot find the optimum for functions f_{15}, f_{19} and f_{20} and gets stuck in a local optimum in the corresponding fitness landscapes. Overall, DE, PSO and RBI find for almost all functions the optimal (or near-optimal) solutions

and produce better results than GA and SA, while GA and SA are on the same level regarding the optimisation of the corresponding low-dimensional functions.

Results for the 30-dimensional Functions

	RBI	DE	PSO	GA	SA	Optimum
F1	0.000000 e+00	0.000000 e+00	0.000000 e+00	1.6828756 e-06	2.390598 e-10	0.000000 e+00
F2	0.000000 e+00	8.142916 e-37	0.000000 e+00	5.466097 e-03	5.140537 e-05	0.000000 e+00
F3	0.000000 e+00	0.000000 e+00	0.000000 e+00	6.52916 e-04	1.29546415 e-05	0.000000 e+00
F4	3.223638 e-15	3.7110183 e-08	3.7296683 e-11	1.9280297 e+00	9.731591 e-01	0.000000 e+00
F5	2.620019 e+01	4.03334 e-22	2.2833145 e+01	3.3183784 e+01	6.4433184 e+00	0.000000 e+00
F6	0.000000 e+00	0.000000 e+00	0.000000 e+00	0.000000 e+00	0.000000 e+00	0.000000 e+00
F7	3.419498 e-04	3.215471 e-03	1.6625043 e-03	8.144887 e-04	3.7914343 e-02	0.000000 e+00
F8	-1.2569486 e+04	-1.2569486 e+04	-1.0557845 e+04	-7.4431504 e+03	-1.2558608 e+04	-1.2569486 e+04
F9	0.000000 e+00	0.000000 e+00	2.527239 e+01	3.0669328 e-04	1.9462983 e-01	0.000000 e+00
F10	8.970602 e-15	3.996803 e-15	7.312669 e-15	9.4749196 e-04	1.9690224 e-04	4.4408 e-16
F11	0.000000 e+00	0.000000 e+00	1.0678519 e-03	8.698255 e-03	5.5567506 e-07	0.000000 e+00
F12	1.5705448 e-32	1.5705448 e-32	3.4549083 e-03	1.2341902 e-08	3.499239 e-10	1.5705448 e-32
F13	-1.1504403 e+00	-1.1504403 e+00	-1.1504403 e+00	-1.1504400 e+00	-1.1504403 e+00	-1.1504403 e+00

Figure 5.5: The averaged best fitness values obtained by DE, PSO, GA, SA and RBI for the high-dimensional functions shown in Fig. 5.3. The functions are implemented in 30 dimensions. Best solutions are shown in grey.

We have a different situation in case of the optimisation of high-dimensional functions. Fig. 5.5 shows the experimental results regarding the optimisation of functions from f_1 to f_{13}, where each of them is implemented in 30 dimensions.

As mentioned above, the functions from f_1 to f_7 have unimodal and the functions from f_8 to f_{13} have multimodal fitness landscapes. f_7 is a noisy function with a unimodal fitness landscape. For functions f_1 - f_4 RBI finds always the optimum, while PSO can find the optimum for f_1, f_2 and f_3 and DE only for f_1 and f_3. Neither GA nor SA can provide the optimal solution for these functions. On f_5 (Rosenbrock function), DE finds the best (near-optimal) solution, while the results obtained by other algorithms are all far away from the optimum. Here, RBI is on the same level as GA and PSO, and SA provides the worst fitness value. On function f_6 (step function), all algorithms find the optimum. On function f_7 (quadric noise), RBI finds the best solution, while the solution found by GA is close to the solution found by

RBI. Apparently, DE, PSO and SA face difficulties with noisy problems.

DE and RBI find always the optimal solution on functions f_8, f_9, f_{11} and f_{12}, while RBI faces some difficulties in the optimisation of f_{10}. In this case, DE and PSO produce better results than RBI, while the solution found by RBI is very close to the solutions obtained by DE and PSO ($< 10^{-14}$). Finally, all algorithms except for GA find the optimal solution on function f_{13}. Overall, our results show that RBI produces better results than PSO, GA, and SA and is on the same level as DE for the given 30-dimensional benchmark functions.

Results for the 50-dimensional Functions

	RBI	DE	PSO	GA	SA	Optimum
F1	0.000000 e+00	1.781570 e-40	0.000000 e+00	8.629046 e-06	3.2896692 e-06	0.000000 e+00
F2	0.000000 e+00	3.399447 e-21	0.000000 e+00	1.7860996 e-02	9.279108 e-03	0.000000 e+00
F3	0.000000 e+00	3.6532754 e-35	0.000000 e+00	1.2131256 e-02	3.8390574 e-01	0.000000 e+00
F4	6.029871 e-07	1.5074507 e-02	3.5901278 e-01	5.8441305 e+00	7.6963463 e+00	0.000000 e+00
F5	5.215521 e+01	1.2438004 e+01	4.2710625 e+01	6.40643 e+01	7.71717 e+01	0.000000 e+00
F6	0.000000 e+00	0.000000 e+00	0.000000 e+00	0.000000 e+00	0.000000 e+00	0.000000 e+00
F7	1.408758 e-03	1.3540829 e-02	8.216297 e-03	4.590097 e-03	1.41383 e-01	0.000000 e+00
F8	-2.088923 e+04	-2.0949145 e+04	-1.637264 e+04	-1.2030071 e+04	-2.0509068 e+04	-2.0949145 e+04
F9	3.169004 e+01	0.000000 e+00	9.195580 e+01	5.3257763 e-01	5.8090005 e+00	0.000000 e+00
F10	1.3707554 e-14	7.5495166 e-15	1.0510111 e-14	1.6838068 e-03	7.125051 e-02	4.4408 e-16
F11	0.000000 e+00	0.000000 e+00	1.0681802 e-03	1.312216 e-02	2.2272845 e-03	0.000000 e+00
F12	9.423269 e-33	9.423269 e-33	1.5821074 e-01	5.1047902 e-08	2.8394586 e-06	9.423269 e-33
F13	-1.1504403 e+00	-1.1504403 e+00	-1.1497078 e+00	-1.1504385 e+00	-1.1503644 e+00	-1.1504403 e+00

Figure 5.6: The averaged best fitness values obtained by DE, PSO, GA, SA and RBI for the high-dimensional functions shown in Fig. 5.3. The functions are implemented in 50 dimensions. Best solutions are shown in grey.

The results in Fig. 5.6 show that RBI and PSO produce the best results for functions f_1 - f_3, while we observe a decrease in the performance of DE for these functions. We have observed that DE requires more than 500,000 function evaluations (approximately between 700,000 and 1,000,000) to find the optimum for the functions f_1 - f_3 showing that the convergence speed of DE decreases significantly, if we increase the number of dimensions. GA and SA perform on functions f_1 - f_3 worse than RBI, PSO and DE providing similar suboptimal solutions. The results on other unimodal functions (f_4 - f_7) show that RBI produces the best results except for f_5. There, in spite of its low convergence speed DE outperforms RBI, PSO, GA and SA and provides

the best result.

On functions f_8 and f_9, we observe a decrease in the performance of RBI. Our experiments on function f_8 have shown that RBI can find in 70% of all experiments the optimum, while it gets stuck in a near-optimum solution in the rest of the experiments. On function f_9, we have observed that RBI can find the optimum only if we increase the maximum number of function evalutions from 500,000 to 1,000,000. The reason is that the number of local minima in the fitness landscape of f_9 increases exponentially, if we increase the number of dimensions from 30 to 50. Here, RBI can avoid the local minima using its exploration scheme, but it requires much time for getting closer to the global optimum due to the complex form of the fitness landscape. The function f_9 with 50 dimensions is the single exceptional case within our benchmark functions, where DE has a higher convergence rate than RBI. For function f_{10}, RBI, DE and PSO produce very similar results, while they again perform better than GA and SA. For functions f_{11}, f_{12} and f_{13}, RBI and DE find always the optimum, while PSO, GA and SA can produce good (but not optimal) results on these functions.

The results in Fig. 5.6 show that RBI and DE are on the same level providing clearly better results than PSO, GA and SA for the functions f_1 - f_{13} with 50 dimensions.

In this section, we have investigated the performance of RBI using noiseless functions (except for f_7). In the next section 5.1.3, we present the noisy benchmark functions and investigate to what extend the algorithms can cope with the existing noise in the environment. Afterwards in Sec. 5.1.4, we will look at the other important criterion for a comparison of optimisation algorithms namely the convergence speed. For the sake of convenience, in the rest of this chapter we concentrate on the optimisation of high-dimensional unimodal and multimodal functions presented in Fig. 5.2, since the low-dimensional functions presented in Fig. 5.3 are trivial to optimise and do not provide a clear conclusive study on the performance of the investigated set of algorithms (see the results in Sec. 5.1.2).

5.1.3 RBI in Noisy Environments

The investigation of noisy functions is of special interest for us, since noise models real-world effects like disturbances, and OC systems are designed to work in environments that involve disturbances [10]. In this context, an OC system must have the capability to adapt to (internal or external) changes and to maintain a required functionality in spite of a certain range of parameter variations. Thus, in this section we investigate to what extent the algorithms are capable to alleviate the negative effect of noise and to find acceptable solutions. The results presented in Fig. 5.5 and in Fig. 5.6 show that RBI can optimise the noisy function f_7 better than its competitors PSO, DE, GA and SA. If we look at the function f_7 again, we see that the fitness value of a solution is manipulated using a uniformly distributed random number between 0 and 1 so that the noisy fitness values are always larger than the real fitness values. Since noise models real-world effects, it is more adequate and realistic to implement the noise so that the noisy fitness values can be larger as well as less than the real fitness values. In order to implement this type noise we do not use a uniformly distributed, but a normal-distributed random number with the mean of μ and the standard deviation of ζ, where ζ is used to determine the level of the noise. This type of noise model is called "Gaussian noise"[2] (see Eq. 5.1).

$$noisyFitness(S) = realFitness(S) + N(\mu, \zeta) \tag{5.1}$$

According to Eq. 5.1, we calculate first the real fitness value of a solution S, and add $N(\mu, \zeta)$ to this real fitness value, where $\mu = 0$ and $\zeta > 0$. In case of f_7, we remove the existing noise implementation and replace it with $N(\mu, \zeta)$. In this context, we investigate the high-dimensional functions presented in Fig. 5.2 each with 30 dimensions. In order to provide a more comprehensive comparison of the algorithms, we consider two different cases, where the functions are implemented (1) using a moderate noise with $\zeta = 1$, and (2) using a severe noise with $\zeta = 2$. The maximum number of function evaluations is set to 500,000 for each algorithm. 30 experiments, each initialised with a different

[2] $http://en.wikipedia.org/wiki/Gaussian\_noise$

random seed, are carried out and the average best fitness values are recorded. Fig. 5.7 and in Fig. 5.8 show the results provided by RBI, DE, PSO, GA and SA for the functions with moderate and severe noise, respectively.

	RBI		DE		PSO		GA		SA		Optimum	
F1	2.6016408	e-01	7.968097	e-01	7.013338	e-01	3.4891313	e-01	3.8507392	e+00	0.000000	e+00
F2	1.0070823	e+00	2.146853	e+00	1.6780392	e+00	1.3022784	e+00	8.874142	e+00	0.000000	e+00
F3	3.685195	e-01	8.237387	e-01	6.219726	e-01	7.5976145	e-01	6.051387	e+00	0.000000	e+00
F4	2.016654	e+00	5.1229944	e+00	5.593701	e+00	1.847722	e+01	4.9850307	e+01	0.000000	e+00
F5	2.7900732	e+01	2.530693	e+01	3.1385082	e+01	4.909949	e+01	5.1053944	e+01	0.000000	e+00
F6	0.000000	e+00	0.000000	e+00	0.000000	e+00	1.3333334	e-01	4.633333	e+00	0.000000	e+00
F7	1.7144777	e-02	2.4342616	e-01	2.0309965	e-01	2.5565637	e-02	1.3845968	e+00	0.000000	e+00
F8	-1.2569094	e+04	-1.2568671	e+04	-1.044732	e+04	-7.7821445	e+03	-1.2544497	e+04	-1.2569486	e+04
F9	9.2712325	e-01	8.285188	e-01	2.9050192	e+01	1.2117425	e+01	1.7366898	e+01	0.000000	e+00
F10	2.46953	e+00	2.0319529	e+01	4.015656	e+00	7.3104186	e+00	2.0414267	e+01	4.4408	e-16
F11	1.3473115	e+00	1.7802763	e+00	1.7256411	e+00	1.2492676	e+01	5.2477098	e+00	0.000000	e+00
F12	1.8701851	e-01	1.3175092	e+00	1.2259672	e+00	3.7458856	e+00	4.719795	e+00	1.5705448	e-32
F13	-6.937364	e-01	-2.0371327	e-01	-2.7448362	e-01	3.3093426	e+00	4.1264434	e+00	-1.1504403	e+00

Figure 5.7: The averaged best fitness values obtained by DE, PSO, EA, SA and RBI for the functions with moderate noise ($\zeta = 1$). Best solutions are shown in grey.

	RBI		DE		PSO		GA		SA		Optimum	
F1	4.73917	e-01	1.6055187	e+00	1.3323575	e+00	7.4745417	e-01	7.679526	e+00	0.000000	e+00
F2	2.1392787	e+00	4.302692	e+00	3.451572	e+00	3.2485104	e+00	1.5587367	e+01	0.000000	e+00
F3	7.460494	e-01	1.5683665	e+00	1.4230508	e+00	1.5301583	e+00	1.2064701	e+01	0.000000	e+00
F4	2.7777069	e+00	1.0254095	e+01	9.610751	e+00	1.9387043	e+01	6.429114	e+01	0.000000	e+00
F5	3.4213844	e+01	2.7154129	e+01	3.197307	e+01	5.634384	e+01	7.5099884	e+01	0.000000	e+00
F6	0.000000	e+00	0.000000	e+00	0.000000	e+00	5.5333333	e+00	1.0200	e+01	0.000000	e+00
F7	3.254504	e-02	4.6489128	e-01	4.1390216	e-01	3.4557793	e-02	2.8130367	e+00	0.000000	e+00
F8	-1.2568687	e+04	-1.2567873	e+04	-1.0345221	e+04	-7.79787	e+03	-1.2553951	e+04	-1.2569486	e+04
F9	9.534879	e+00	6.3235893	e+00	3.235611	e+01	1.2542366	e+01	2.9559116	e+01	0.000000	e+00
F10	3.871318	e+00	2.0548817	e+01	1.8824408	e+01	8.678788	e+00	2.0570177	e+01	4.4408	e-16
F11	1.6238683	e+00	2.5910337	e+00	2.345884	e+00	1.7346428	e+01	9.443265	e+00	0.000000	e+00
F12	4.1231167	e-01	2.6070645	e+00	2.1659403	e+00	5.051514	e+00	7.56975	e+00	1.5705448	e-32
F13	-1.8783651	e-01	7.61761	e-01	6.673315	e-01	8.098681	e+00	9.138774	e+00	-1.1504403	e+00

Figure 5.8: The averaged best fitness values obtained by DE, PSO, EA, SA and RBI for the functions with severe noise ($\zeta = 2$). Best solutions are shown in grey.

The results in Fig. 5.7 and in Fig. 5.8 show clearly that RBI produces the best results for the majority of the functions both with moderate and severe noise. Generally, if we optimise a noisy function using a population-based optimisation algorithm, there exist two types of agents in the population: (1) Agents that are *strongly* affected by the noise, and (2) agents that are *slightly* affected by the noise. In this context, RBI alleviates the negative

effect of the noise in that the RBI-agents consider not only a single agent but a group of agents (comparison set) in determining the new solutions. Hence, the comparison set of a particular agent consists of both types of agents, which makes it possible for RBI to adjust its search towards the optimum using the agents that are only slightly affected by the noise. In order to explain this aspect, we assume that an agent is only slightly affected by the noise, if the noise component $N(\mu, \zeta)$ presented in Eq. 5.1 produces a value between [-0.5, +0.5]. According to the Gaussian distribution with $\mu = 0$ and $\zeta = 1$, the noise value is between [-0.5, +0.5] with the probability of 38.29% ($\approx$ 40%) [3]. Since, each RBI-agent has 10 randomly selected agents in its comparison set (see the parameter settings in Sec. 5.1.1), there exist approximately 4 agents in each comparison set that are only slightly affected by the noise. This makes it for an RBI-agent possible to adjust its search towards the optimal or near-optimal solution. Nevertheless, the population requires very long time (>> 500,000 function evaluations) to determine the exact optimum, since the agents that are strongly affected by the noise redirect the search away from the optimum and thus permanently slow down the optimisation process.

The exploration behaviour of RBI-agents in combination with the use of the comparison set in determining new solutions results in an effective optimisation scheme for the noisy functions. In this context, RBI does not switch from more exploration to more exploitation towards the end of the optimisation. Rather, RBI-agents always explore the fitness landscape according to the RBI-scheme presented in 4.1 even after the population has converged to a specific area in the search space.

If we look at the results presented in Fig. 5.7 in more detail, we see that PSO is the second best performing algorithm after RBI. PSO provides better results than DE, GA and SA in the majority of the investigated functions. GA produces better results than PSO only for functions f_1, f_2, f_7 and f_9, SA for functions f_8 and f_9 and DE for functions f_4, f_5, f_8 and f_9. The results show that GA, DE and SA face difficulties in the optimisation of noisy functions, since they switch from more exploration to more exploitation towards the end of the optimisation, which results in a premature convergence. In this context,

[3] http://www.stat.wvu.edu/SRS/Modules/Normal/normal.html

PSO (as well as RBI) follows the more successful exploration pattern, where the particles (agents) continue exploring the fitness landscape even after the population has converged to a specific part of the search space. Each particle in PSO is propelled towards its own best position and the best position found so far in its neighbourhood. Thus, the particles in PSO use only two solutions, while they optimise. The agents in RBI, on the other hand, optimise based on all solutions represented by the agents in the comparison set, which gives the RBI-agents the capability to adjust their search towards the optimal solution more effectively than the particles in PSO.

The results presented in Fig. 5.8 regarding the functions with severe noise look similar to the results presented in Fig. 5.7. Again, the successful exploration behaviour of the particles in PSO leads to better results than the results obtained by GA, DE and SA. Thus, PSO is the second best performing algorithm after RBI, while GA and DE perform more or less on the same level. The results show that all algorithms provide better results than SA in the majority of the investigated functions. Overall, the results show that RBI can perform effectively in noisy environments and outperforms PSO, DE, GA and SA. Thus, RBI is more suitable than its competitors to be used for OC systems that work in noisy environments[4].

5.1.4 Convergence Speed of RBI

In the previous sections, we have investigated the first important criterion for the comparison of optimisation algorithms, which is the quality of solutions provided in noiseless and noisy environments. Generally, it is not only important to determine how good the solutions are, but also how many iterations are required to find the corresponding solutions. Thus, we deal now with the second important comparison criterion in this section, which is the convergence speed. Principally, the convergence speed of an optimisation algorithms is measured using the *number of function evaluations* ($\#FE$) required to produce a certain function value (i.e., a success criterion)[132, 99]. Thus, in order to determine a convergence speed of an algorithm, we have to define a suc-

[4]Further experimental results on noisy functions can be found in the Appendix on page 144.

cess criterion for each considered function. In this context, we implemented the functions in Fig. 5.2 (high-dimensional functions) in 30 dimensions and defined for each function a proper success criterion so that the algorithms can satisfy each corresponding criterion requiring no more than 500,000 function evaluations. Each function is optimised 30 times, and the average values are recorded. Fig. 5.9 shows the success criterion defined for each function together with the corresponding optimum, and Fig. 5.10 shows the number of function evaluations required for each algorithm to achieve the success criterion given in Fig. 5.9.

	Success criterion	Optimum
F1	$F_{best\_1} < 0.01$	0.0000000 e+00
F2	$F_{best\_2} < 0.01$	0.0000000 e+00
F3	$F_{best\_3} < 0.01$	0.0000000 e+00
F4	$F_{best\_4} < 5$	0.0000000 e+00
F5	$F_{best\_5} < 100$	0.0000000 e+00
F6	$F_{best\_6} < 1$	0.0000000 e+00
F7	$F_{best\_7} < 0.1$	0.0000000 e+00
F8	$F_{best\_8} < -6500$	-1.2569486 e+04
F9	$F_{best\_9} < 100$	0.0000000 e+00
F10	$F_{best\_10} < 2.5$	4.4408 e-16
F11	$F_{best\_11} < 0.1$	0.0000000 e+00
F12	$F_{best\_12} < 0.2$	1.5705448 e-32
F13	$F_{best\_13} < 0$	-1.1504403 e+00

Figure 5.9: The success criteria defined for the functions together with their optima. F_{best_i} represents the fitness value that should be achieved by an algorithm to satisfy the success criterion for the function i.

The results presented in Fig. 5.10 show clearly that RBI has the highest convergence speed for all considered functions except for f_9, where PSO and GA perform better. SA, on the other hand, is the worst performing algorithm with the lowest convergence speed. Only for function f_8, SA provides a good result performing better than GA and DE. Based on these results, RBI is the best and SA is the worst performing algorithm regarding the measured convergence speeds. PSO produces good results especially for functions f_2, f_3,

#FE	RBI	DE	PSO	GA	SA
F1	9750.00	29640.00	14826.66	12800.00	151601.67
F2	20546.66	54993.33	28173.33	296393.34	281790.84
F3	29106.66	63510.00	34376.66	161586.67	344800.84
F4	6230.00	64683.33	29533.33	149236.67	248167.50
F5	12253.33	36496.66	23623.33	35003.33	197495.00
F6	10973.33	32563.33	18430.00	17343.33	172253.33
F7	1836.66	22916.66	6840.00	2680.00	198515.83
F8	4150.00	12233.33	6240.00	50790.00	8122.50
F9	19626.66	28873.33	17023.33	4743.33	66582.50
F10	6873.33	26440.00	12470.00	8963.33	178097.50
F11	17406.66	45456.66	23443.33	54460.00	218437.50
F12	6853.33	34973.33	34746.66	58576.66	136327.50
F13	7510.00	30746.66	19363.33	14510.00	137881.67

Figure 5.10: The convergence speed of RBI, DE, PSO, GA and SA measured in terms of the number of function evaluations ($\#FE$) required to achieve the success criteria given in Fig. 5.9. Best results are shown in grey.

f_6, f_8 and f_{11}. In this case, PSO achieves the corresponding success criteria requiring almost the same number of function evaluations as RBI, while the convergence speed of PSO is higher than the convergence speed of DE and SA for these functions. PSO outperforms also GA for functions f_2, f_3, f_4, f_5, f_8, f_{11} and f_{12}, while GA is only slightly better than PSO in the rest of the benchmark functions. Thus, PSO is the second fastest algorithm after RBI according to our benchmark functions. Furthermore, the results show that GA outperforms DE in 7 out of 13 functions so that both algorithms are more or less on the same level regarding their convergence speed. Overall, RBI is the best performing algorithm followed by PSO, GA - DE, and SA in the order of the convergence speed.

Although, it is common to use the number of function evaluations to measure the convergence speed, we have also measured the convergence speed by the amount of time in milliseconds required to achieve the intended function values. Here, all experiments have been carried out on the same machine with an Intel 1.8 dual core CPU, 2 GB memory and Windows XP operating system. Fig. 5.11 shows the time in milliseconds required for each algorithm to achieve the success criteria given in Fig. 5.9

	RBI	DE	PSO	GA	SA
F1	206.26 ms.	673.96 ms.	247.90 ms.	280.73 ms.	3718.30 ms.
F2	363.03 ms.	910.86 ms.	385.36 ms.	5055.90 ms.	5386.56 ms.
F3	721.40 ms.	1239.10 ms.	731.70 ms.	3905.70 ms.	10384.03 ms.
F4	109.36 ms.	930.20 ms.	377.03 ms.	2586.23 ms.	4747.50 ms.
F5	217.76 ms.	945.30 ms.	553.73 ms.	609.93 ms.	3824.03 ms.
F6	272.40 ms.	757.36 ms.	367.20 ms.	433.40 ms.	5022.50 ms.
F7	96.90 ms.	1307.20 ms.	318.23 ms.	146.36 ms.	13300.66 ms.
F8	113.53 ms.	498.93 ms.	142.23 ms.	1374.60 ms.	277.06 ms.
F9	548.96 ms.	888.50 ms.	437.50 ms.	149.46 ms.	2552.06 ms.
F10	220.86 ms.	881.80 ms.	352.10 ms.	288.56 ms.	7466.26 ms.
F11	518.80 ms.	1232.86 ms.	604.20 ms.	1633.46 ms.	8534.46 ms.
F12	265.06 ms.	1384.43 ms.	1156.76 ms.	957.86 ms.	8684.46 ms.
F13	319.73 ms.	1462.50 ms.	825.00 ms.	555.80 ms.	9864.70 ms.

Figure 5.11: The convergence speed of RBI, DE, PSO, GA and SA in milliseconds regarding the success criteria given in Fig. 5.9. Best results are shown in grey.

The results presented in Fig. 5.11 correlate with the results presented in Fig. 5.10. Here, RBI is again the fastest algorithm for all benchmark functions except for f_9. At this point, it is important to note that the convergence speed measured in ms depends highly on the evaluation mechanism as mentioned in Sec. 2.3.3. In our experiments, the cost of the evaluation mechanism is very small, since we just have to calculate a function in order to determine the fitness of a solution. In different scenarios, such as in OTC [33, 24] or in ONC [133], the time required for the evaluation of a particular solution can be very high (>>100 ms), if we use e.g., a simulator to determine the fitness values instead of calculating a simple function. In such a case, the convergence speeds of the algorithms would differ significantly according to (1) the evaluation mechanism used and (2) the number of function evaluations required to achieve the success criterion. Nevertheless, the results presented in Fig. 5.11 give an idea about how much time the algorithms require to provide acceptable results regarding the considered benchmark functions.

5.1.5 Conclusion

In this section, we have investigated the performance of RBI in comparison to DE, PSO, GA and SA using noiseless and noisy functions. Furthermore, we have measured the convergence speed of the considered algorithms in terms of the number of function evaluations required to achieve the predefined success criteria. In this context, we have also shown experimental results on the convergence speed of the algorithms regarding the required time measured in milliseconds. The results for the noiseless functions are summarised in Fig. 5.12.

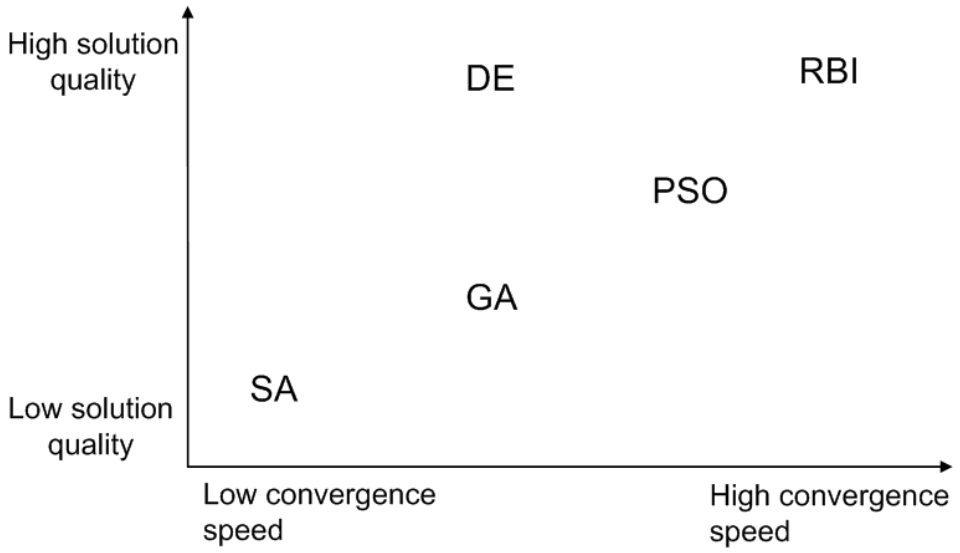


Figure 5.12: The comparison of convergence speeds and the quality of solutions provided by RBI, DE, PSO, GA and SA in case of noiseless functions.

As shown in Fig. 5.12, RBI is the best performing algorithm with the highest convergence speed and the best quality of solutions. Regarding the quality of solutions, DE performs just as well as RBI, while its convergence speed is lower than RBI. The quality of solutions produced by PSO is lower than DE and RBI (especially in case of the 50-dimensional functions), while the convergence speed of PSO is higher than the convergence speeds of DE, GA and SA. GA provides solutions that are better than the solutions produced by SA and worse than those found by RBI, DE, and PSO. The convergence speed of GA is as high as the convergence speed of DE and lower than the

convergence speed of PSO and RBI. Our experiments have shown that SA is the algorithm with the worst solution quality and the lowest convergence speed within the set of investigated algorithms.

The investigation of the noisy functions shows that there is a change in the relative order of the algorithms regarding the quality of solutions (see Fig. 5.12).

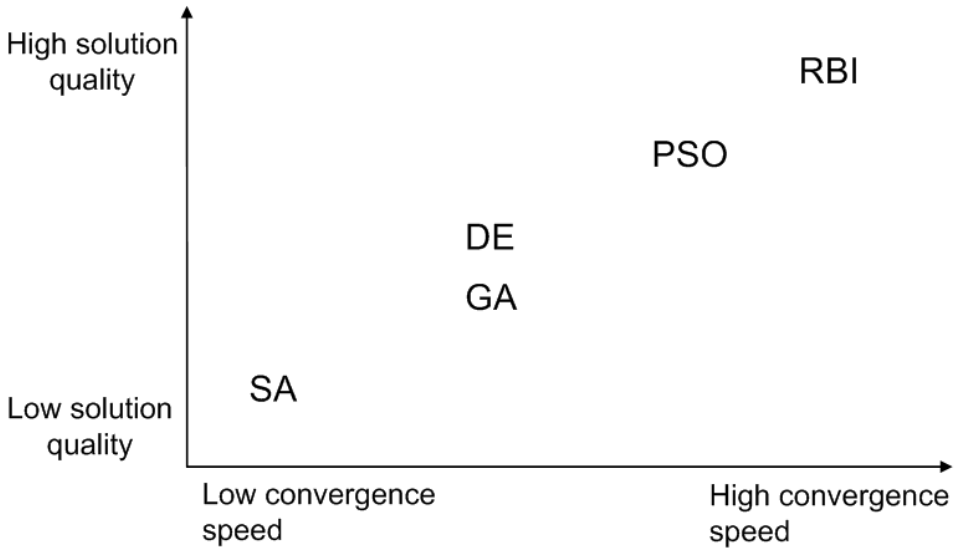


Figure 5.13: The comparison of convergence speeds and the quality of solutions provided by RBI, DE, PSO, GA and SA in case of noisy functions.

In this case, we have observed that the performance of DE decreases drastically so that it produces worse results than PSO. Thus, in this case RBI is the best and PSO is the second best performing algorithm followed by DE and GA, while the results obtained by DE are slightly better than the results obtained by GA. Here, SA has provided again the worst results regarding both the convergence speed and the quality of solutions. Overall, RBI produces the best results and is more suitable than its competitors to be used for OC systems that work in noisy environments.

5.2 Combinatorial Optimisation with RBI

In this section, we investigate the performance of RBI in discrete search spaces using the Traveling Salesman Problem (TSP). Here, we compare RBI to the optimisation algorithms Simulated Annealing (SA), Genetic Algorithm (GA), Ant Colony Optimisation (ACO), Evolutionary Programming (EP) and Annealing Genetic Algorithm (AG) [134]. In this context, we have implemented topologies with different numbers of cities in order to determine the performance of the algorithms according to (1) the quality of solutions obtained and (2) the converge speed measured in terms of function evaluations. Here, we put special emphasis on the comparison of ACO and RBI, since ACO is particularly developed to solve shortest path problems with a graph representation such as TSP. Thus, our investigation in this section includes a more detailed comparison of RBI and ACO. In the next section 5.2.1, we discuss the parameter settings we have used for the algorithms followed by the experimental results presented in Sec. 5.2.2.

5.2.1 Parameter Settings

The benchmark problems as well as the results that we have used to compare RBI to AG, EP, SA and GA are taken from [134]. Thus, we have not implemented AG, EP, SA and GA for TSP, instead we use the corresponding results directly from [134]. A more detailed comparison is provided regarding RBI and ACO. In this context, we have implemented RBI and ACO for TSP, and carried out an intensive parameter study to determine the best configuration for them.

RBI

In order to determine an adequate configuration for RBI, we have carried out a parameter study using different TSPs from the TSPLIB[5], which is a collection of different benchmark problems with varying difficulties. Accordingly, we set the parameter α, which is used to determine the convergence of solutions in

[5]http://comopt.ifi.uni-heidelberg.de/software/TSPLIB95/

the comparison set, to $1/10^{th}$ of the number of cities in the given TSP. For example, we set α to 10, if we optimise a TSP with 100 cities. The second parameter β, which is used to select one of either the inductive exploration or the exploitation steps, is set to 1.5 (see Fig. 4.4 for α and β). Furthermore, the population and the neighbourhood sizes are set to 20 and 5, respectively.

ACO

ACO has been extensively investigated in the literature [30]. We have obtained the best results with ACO using the Ant Colony System (ACS), which is a particular instance of ACO, with the parameter setting proposed in [30] on page 71, where $\alpha = 1$, $\beta = 3$, $\rho = 0.1$, $m = 10$ and $\tau_0 = 1/nC^{nn}$.

5.2.2 Solving TSP using RBI

In order to investigate the performance of RBI, we have implemented four topologies with 30 (Oliver30), 50 (Elion50), 75 (Eilon75) and 100 (KroA100) cities, which are investigated in [134] in detail. There, Dorigo et al. compare ACO to GA, EP, SA and AG using the these four TSPs and present the best integer tour length obtained by each algorithm together with the number of tours (this corresponds to the number of function evaluations from Sec. 5.1.4) required to find this tour length. We use these results from [134] and compare them with the results of RBI. We have carried out 10 experiments for each TSP, where the maximum number of function evaluations in each experiment is set to 150,000. Fig. 5.14 shows the experimental results.

According to the results in Fig. 5.14, ACO is the single algorithm that finds the optimum for each TSP requiring a very small number of function evaluations. RBI can find the optimum only for Oliver30, Eilon50 and Eilon75 requiring a large number of function evaluations in comparison to ACO. In none of the 10 experiments RBI has managed to find the optimum for KroA100. GA and EP perform more or less on the same level for the given TSPs providing comparable results that are both worse than the results produced by RBI. At this point, we emphasise again that RBI utilises genetic operators (i.e., crossover and mutation) such as GA in order to optimise in discrete search

	Oliver30		Eilon50		Eilon75		KroA100	
	Tour length	NFE	Tour length	NFE	Tour length	NFE	Tour length	NFE
ACS	420	830	425	1830	535	3480	21282	4820
RBI	420	3760	425	20820	535	34920	21445	65160
GA	421	3200	428	25000	545	80000	21761	103000
EP	420	40000	426	100000	542	325000	NA	NA
SA	424	24617	443	68512	580	173250	NA	NA
AG	420	12620	436	28111	561	95506	NA	NA
Optimum	420		425		535		12282	

Figure 5.14: The comparison of ACO, RBI, GA, EP, SA and AG for the TSPs with 30, 50, 75 and 100 cities. The results show the best integer tour length and the number of function evaluations (NFE) required to find the corresponding tour length. NA stands for "Not available", since there are no known results for EP, SA and AG regarding the 100-city problem KroA100. Best results are shown in grey.

spaces, while the exploring and exploiting agents in RBI are determined according to the RBI-scheme instead of using some selection mechanisms, which are specific to GA (see Sec. 3.3.1). Thus, although RBI and GA both use genetic operators, the utilisation of the RBI-scheme leads to an improvement of the solution quality and the convergence speed in comparison to GA. The remaining results show that AG performs worse than GA and EP, and better than SA. Hence, SA is the worst performing algorithm regarding the given benchmark TSPs. Overall, ACO is the best performing algorithm followed by RBI, EP, GA, AG and SA.

The results in Fig. 5.14 show that RBI and ACO are the two best performing algorithms regarding the considered set of TSPs suggesting further investigation of them. In this context, we extend the benchmark problems Oliver30, Eilon50, Eilon75 and KroA100 with additional TSPs from TSPLIB together with randomly created TSPs of different sizes. Here, we carry out 10 experiments for each TSP, where the maximum number of function evaluations is set to 150,000. Fig. 5.15 shows the average tour length and the average number of function evaluations obtained by ACO and RBI for each TSP.

The results in Fig. 5.15 confirm the results presented in Fig. 5.14. Again, ACO performs better than RBI in the majority of the investigated problems. RBI produces better results than ACO in TSPs with 25, 150, and 200 cities,

	RBI		ACO		Optimum
Number of cities	Tour length	NFE	Tour length	NFE	
25 (random)	20339.6	17516	20352.4	16816	20274
30 (Oliver30)	420	25888	420	1068	420
40 (random)	27092	8532	27092	10040	27092
50 (Eilon50)	426.22	35984	425.8	53789	425
50 (random)	29135	28258	29135	5837	29135
52 (Berlin52)	7542	24480	7542	5622	7542
65 (random)	32717.2	75280	32642.7	72116	32549
70 (St70)	683.8	91338	680.9	60677	675
75 (Eilon75)	537.8	79900	535	23580	535
99 (Rat99)	1220.3	68720	1216.7	63731	1211
100 (KroA100)	21486.4	45178	21296.3	78569	21282
100 (KroB100)	22374.4	115182	22272.6	55224	22141
100 (KroC100)	20960.5	80442	20820.2	44286	20749
150 (random)	48609.7	118698	49190.9	42056	47981
159 (U159)	43517.7	73216	42211	65198	42080
200 (KroA200)	30349.7	102530	31082.1	48007	29368
200 (KroB200)	31193.3	100656	31604.6	48456	29437

Figure 5.15: The comparison of ACO and RBI using TSPs of different sizes. The results show the average tour length and the average number of function evaluations (NFE) required to find the corresponding tour length. Best results are shown in grey.

while both algorithms produce the same results for the TSPs with 30, 40, 50 and 52 cities. However, if we look at the quality of solutions produced by the algorithms, we see that ACO is just slightly better than RBI so that the results obtained by ACO and RBI are very similar to each other. The situation looks different if we consider the number of function evaluations required to produce the corresponding results. In this case, ACO has a higher convergence speed than RBI and requires a smaller number of function evaluations to achieve the corresponding results. RBI is faster than ACO only in TSPs with 40 and 50 cities, while the convergence speeds of both algorithms are comparable in TSPs with 25, 65, 99 and 159 cities.

Overall, the results in Fig.5.15 show that ACO performs (slightly) better than RBI regarding the quality of solutions, while its convergence speed is higher than the convergence speed of RBI.

5.2.3 Conclusion

The results presented in Sec. 5.2.2 are summarised in Fig. 5.16.

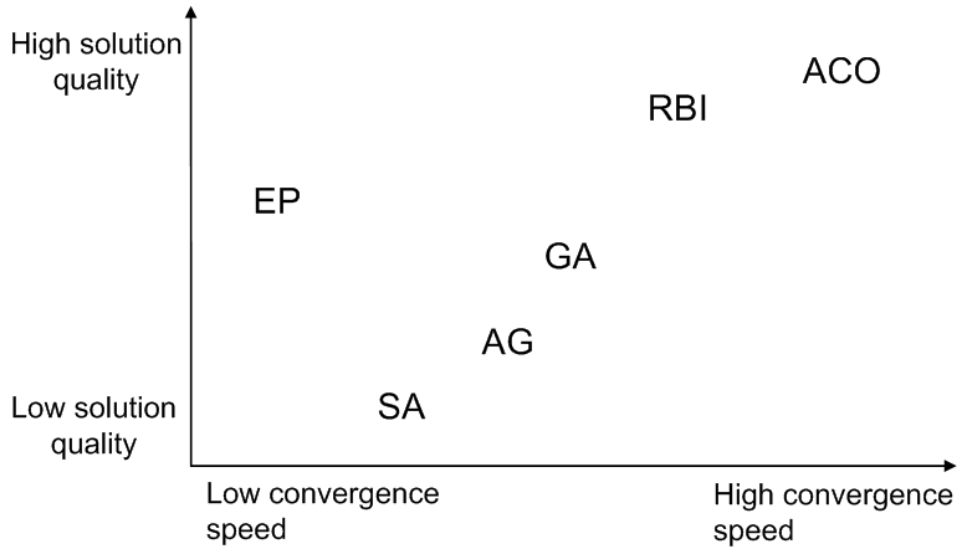


Figure 5.16: The comparison of convergence speeds and the quality of solutions produced by RBI, ACO, GA, SA and AG for the Traveling Salesman Problem.

Although the results obtained by ACO are better than the ones obtained by RBI, there are some restrictions regarding the optimisation using ACO:

1) ACO can only be used to solve discrete optimisation problems that have a graph representation, since it is particularly developed to find the shortest path in a given graph using the environment as a communication medium (stigmergy). On the other hand, RBI is applicable to all kind of discrete and continuous optimisation problems.

2) OC systems have typically self-referential fitness landscapes, where the form of the corresponding fitness landscape changes as a function of the agent behaviour. In case of TSP, this would mean that the ants change the topology of the given TSP by changing their routes. Thus, there would be a new shortest path (permutation of cities) after each optimisation step preventing the ants to increase the pheromone density on the "shortest path", since this "shortest path" changes continuously. Hence, the pheromones would not be deployed

on a specific path, rather they were distributed (possibly) on the whole graph. So, ACO would face difficulties in terms of convergence in self-referential fitness landscapes. RBI, on the other hand, overcomes such problems using its effective exploration/exploitation scheme. RBI-agents do not require a kind of autocatalytic feedback process to converge such as ants, rather they converge in that they keep already found good solutions obtained so far, and at the same time execute the exploitation (imitation) step to move towards these solutions in the corresponding search space (see "Do nothing" and "Imitation" in Sec. 4.3). This type of exploitation guarantees the convergence of the population to good (possibly optimal) solutions, while the remaining agents explore the fitness landscape by executing the random and inductive exploration steps (see "Random exploration" and "Inductive exploration" in Sec. 4.3). Hence, RBI provides a suitable optimisation scheme also for dynamic and self-referential fitness landscapes by (1) finding a (possibly suboptimal) solution at the beginning of the optimisation and (2) *improving* this solution iteratively in the course of the optimisation.

5.3 Summary

In this chapter, we have investigated the performance of RBI in static fitness landscapes. In this context, we have shown experimental results on the performance of RBI in continuous and discrete search spaces using different benchmark functions from the domain of function optimisation and the Traveling Salesman Problem, respectively. In case of the optimisation in continuous search spaces, we have compared RBI to the well-known state-of-the art optimisation algorithms DE, PSO, GA and SA. The results have shown that RBI outperforms all its competitors providing high-quality results (especially in noisy functions) in a short time. Thus, RBI is clearly the best performing algorithm within the set of investigated algorithms for the optimisation of functions defined over continuous variables. In case of the optimisation in discrete search spaces, we have compared RBI to ACO, EP, GA, SA and AG. The results have shown that ACO, which is specially developed to optimise combinatorial problems such as TSP, is the best performing algorithm, while

the results of RBI are very similar to the results obtained by ACO. Hence, these two algorithms are more or less on the same level regarding the quality of solutions and outperform all their competitors. The results regarding the convergence speed show that ACO is the fastest algorithm within the set of investigated algorithms followed by RBI. Overall, the results presented in this chapter give the clear proof that the dynamic role assignment strategy used in RBI facilitates a very effective optimisation scheme guaranteeing high-quality solutions with high convergence speeds in static fitness landscapes.

Chapter 6

Optimisation in Self-referential Fitness Landscapes

In this chapter, we investigate the performance of RBI in dynamic and self-referential fitness landscapes, where the form of the fitness landscape changes as a function of agent behaviour. In this context, we investigate the multi-robot observation scenario from the pursuit (predator/prey) domain [25, 135]. Generally, in a predator/prey scenario the success of a predator does not only depend on its own behaviour, but also on the behaviour of other predators in the system. Thus, predator/prey scenarios provide a good test-bed to investigate the performance of different optimisation algorithms in a self-referential fitness landscape. Furthermore, predator/prey scenarios represent a generic model/abstraction for many multi-agent systems (MAS), and especially for robotic MAS, since they involve agents, which move around in an environment and interact with each other in order to accomplish a given task [136]. In this context, the multi-robot observation scenario involves robots (predators), which work together in order to *collectively* follow and observe the target (prey). This collective behaviour should be learnt in a self-referential fitness landscape, which presents a great challenge for the corresponding optimisation algorithms. Here, we use the Role-based Imitation algorithm (RBI), Differential Evolution (DE), Particle Swarm Optimisation (PSO), Simulated Annealing (SA) and Genetic Algorithm (GA) to learn the optimal collective behaviour

for the robots, and compare the resulting system performances obtained by them. As comparison criteria, we use the convergence speed and the quality of solutions provided by the algorithms as in Chapter 5. Furthermore, we investigate the system performance in the presence of *disturbances* and provide a quantified definition of *robustness* for OC systems using the notion of state spaces as proposed in [40, 10]. We present experimental results regarding the level of robustness obtained by the algorithms.

In Sec. 6.1, we discuss the experimental setup of the multi-robot observation scenario in detail. Sec. 6.2 provides the parameter settings of the algorithms and demonstrates the experimental results for the converge speed and the quality of solutions obtained by each algorithm in scenarios without and with disturbances. In Sec. 6.3, we present a quantified definition of robustness for OC systems and provide experimental results regarding the level of robustness obtained by the algorithms. Sec. 6.4 summarises this chapter.

6.1 Multi-robot Observation Scenario

We use the agent-based modeling and simulation toolkit *RePast* [137] to implement our scenario. RePast provides a scheduler, which triggers agents to perform their predefined behaviour at each time step. A single time step in a RePast simulation is called a *tick*. In our experiments, we also use the notion of ticks in producing experimental results presented in Sec. 6.2. Our scenario consists of robots (predators), which follow the target (prey) in order to *observe* it, while the target tries to evade the robots. We use the terms "robots" for the predators and "target" for the prey in the rest of this chapter. Each robot has an internal variable "number of observations" (abbr. NofOBS), which is incremented at each time step the target is within the observation horizon of the robot. This observation horizon is defined as the 1-step neighbourhood of the robot (see Fig. 6.1).

A robot's local objective is to maximise the value of its internal variable NofOBS, whereas the target tries to evade the robots in order to stay unobserved as often as possible. Using the notion of NofOBS, we measure the system performance as shown in Eq. 6.1.

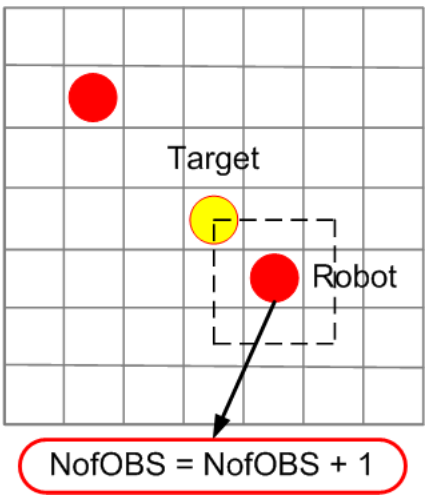


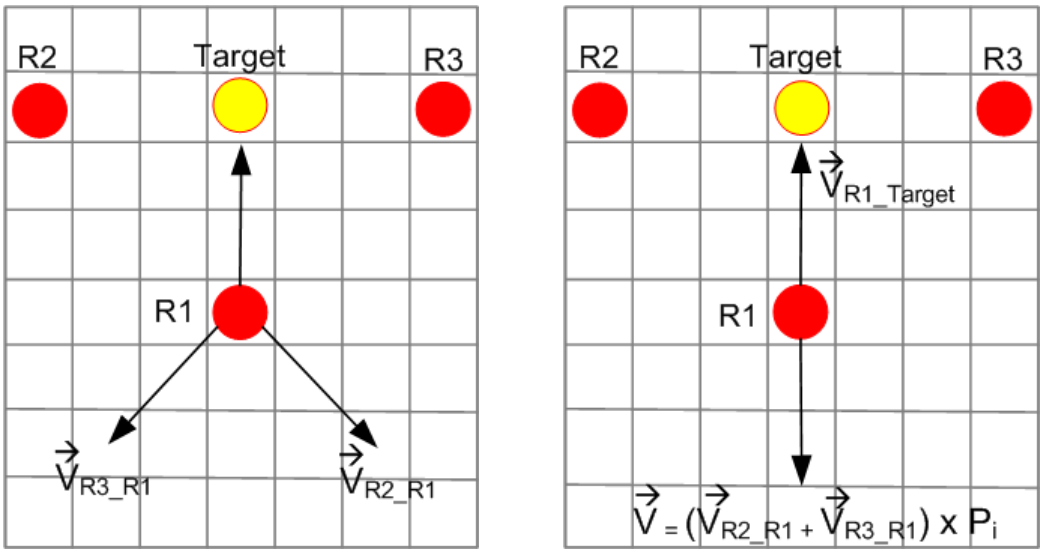
Figure 6.1: A robot increments its NofOBS each time the target is in its observation horizon.

$$Performance(S) = \sum_{i=1}^{m} \zeta(i). \tag{6.1}$$

where m is the number of robots in the system and ζ (i) is the NofOBS of the robot with the index i.

The positions and the moving behaviour of the robots and the target are determined according to an attraction/repulsion (pull/push) model as proposed by Korf [138] together with a hard constraint, which prevents a robot to move into the observation horizon of another robot. Thus, at any given point of time in the simulation, the robots have a minimum of one cell distance between each other so that the target can move continuously (i.e., the target cannot be captured as in a typical predator/prey scenario). Hence, in conflict situations, where e.g., the attraction/repulsion forces of a robot ask it to move into the observation horizon of some other robot, the hard constraint overrides the attraction/repulsion force and the robot does not move. Hence, the local objective of each robot is not to capture the target (since it is not possible in our scenario due to the hard constraint mentioned above), but to maximise the value of its internal variable NofOBS. Fig. 6.2 shows that the robot R1 is attracted by the target and repelled by other robots R2 and R3.

The sizes of the attraction and repulsion vectors are determined by two



(a) The attraction vector towards the target and the repulsion vectors from R2 and R3

(b) The attraction vector and the summation of both repulsion vectors

Figure 6.2: The repulsion and attraction vectors of a robot.

attributes: (1) the distance between the agents and (2) an optimisation parameter P_i used by the robots to adapt the repulsion forces. The magnitude of the repulsion force between two robots (R1 and R2) is calculated as shown in Eq. 6.2:

$$\left\|\vec{V}_{R2\_R1}\right\| = \frac{1}{d(R2, R1)} \tag{6.2}$$

where d(R2, R1) is the distance between R2 and R1. Each robot R in the system determines the attraction vector towards the target and the repulsion vectors from other robots, and calculates the sum of these vectors according to Eq. 6.3:

$$\vec{V}_R = P_i \times \sum_{i=1}^{n} \vec{V}_{Ri\_R} + \vec{V}_{R\_Target} \tag{6.3}$$

where n is the number of robots in the system other than R, and P_i is the repulsion parameter that is used by R to modify the size of the repulsion vectors from other robots (see Fig. 6.2(b)).

The behaviour of the target is determined similarly to the behaviour of a robot. The target has a constant behaviour, i.e. that there is no parameter, which the target can use to change its behaviour over time. The target is repelled by the robots and also from the edges of the grid in order to stay unobserved as often as possible as shown in Fig. 6.3.

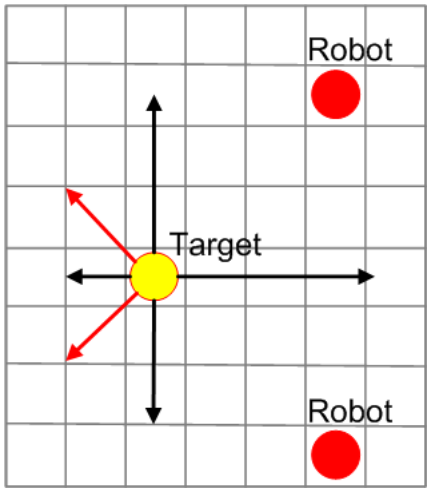


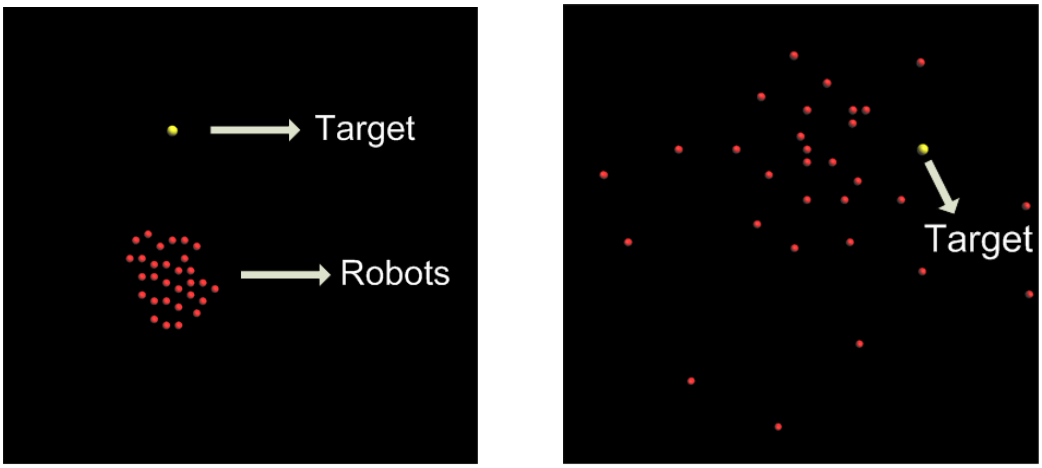
Figure 6.3: The repulsion vectors that determine the behaviour of the target. The repulsion vectors from the robots are shown in red, and the repulsion vectors from the edges of the environment are shown in black.

According to the vectors in Fig. 6.3, the position of the target for the next tick is determined using Eq. 6.4:

$$\vec{V}_{Target} = \sum_{i=1}^{m} \vec{V}_{Ri\_Target} + \sum_{i=1}^{4} \vec{V}_{Edge_i\_Target} \tag{6.4}$$

where m is the number of all robots in the system and $\vec{V}_{Edge_i\_Target}$ is the repulsion vector from the edge with index i. (Notice that the size of each repulsion vector $\vec{V}_{Edge_i\_Target}$ from "Edge$_i$" is calculated using Eq. (6.2)). Furthermore, the target is two times faster than each robot in the system: A robot R can only make one move in a single tick, whereas the target can move twice in the same tick (by calculating $\vec{V}_{Target}$ and moving towards it twice at each tick). Thus, the robots should optimise their repulsion parameters P_i to be able to observe the target collectively and increase the system perfor-

mance. The necessity for an effective optimisation becomes more apparent, if we consider the case, where all repulsion parameters are initialised with 0. In this case, each robot sees only the attraction vector towards the target, and none of them can increase its NofOBS, if they do not optimise their repulsion parameters. This produces a non-collective system behaviour as shown in Fig. 6.4(a), where the robots form a large cluster and follow the target without observing it, since the target moves faster than the robots. Hence, the optimisation of the repulsion parameters allows robots to spatially distribute over the environment and pursue the target collectively (see Fig. 6.4(b)).



(a) The non-collective system behaviour without optimisation where all repulsion parameters (P_i) are set to 0.

(b) The collective system behaviour with optimisation, where the robots spatially distribute over the environment and pursue the target together.

Figure 6.4: The system behaviour with optimising and non-optimising agents where all P_i's are set to 0.

The optimal system performance depends on the repulsion parameters P_i of the robots leading to the question of how these P_i values can be optimised that produce the best system performance. In this context, we use the optimisation algorithms RBI, DE, GA, PSO and SA to determine the optimal repulsion parameters P_i in four different scenarios with an increasing level of complexity as shown in Fig. 6.5.

In all four scenarios presented in Fig. 6.5 the robots try to observe a single target on a grid environment with 70x70 cells. The P_i values are initialised

	Number of robots	Disturbance
Scenario 1	30	No
Scenario 2	50	No
Scenario 3	30	Yes
Scenario 4	50	Yes

Figure 6.5: The scenarios used to compare the optimisation algorithms RBI, DE, GA, PSO and SA.

randomly in the corresponding search space, which is defined between the upper bound 10 and the lower bound -10. We measure the system performance obtained by each algorithm using Eq. 6.1. In this context, we investigate the total number of observations achieved by each algorithm after 50,000 ticks. Each robot in the system calculates its fitness value every 100 ticks according to the following function 6.5:

$$Fitness(R) = \Theta_{t_i}(R) - \Theta_{t_i-100}(R) \tag{6.5}$$

where Θ_{t_i} (R) is the NofOBS of R in the **current** optimisation step at time t_i and $\Theta_{t_{i-100}}$ (R) is the NofOBS of R in the **previous** optimisation step at time t_{i-100}. Since the robots optimise their behaviour every 100 ticks and we measure the system performance after 50,000 ticks, the number of function evaluations ($\#FE$) for a single robot is limited to 500 (50,000 / 100).

The first two scenarios (1-2) do not involve disturbances while the last two scenarios (3-4) do so. In order to create a disturbance in Scenario 3 and Scenario 4, we periodically set the repulsion parameters P_i of the robots to a random value in the search space [-10, 10] so that the system performance decreases until the robots have adapted their repulsion parameters to compensate the disturbance. It is clear that the scenarios with disturbances are more complex than the scenarios without disturbances, since the corresponding optimisation algorithm must not only cope with the self-referential nature of the fitness landscape, but also with the fact that the robots loose their learnt knowledge about the environment periodically. Furthermore, if we look at the scenarios with and without disturbances separately, we see that in each case

a large number of robots result in a more complex scenario, since there are more elements (robots) which influence the form of the corresponding fitness landscape. Thus, an optimisation algorithm can benefit from the large population size only if it can cope with this complexity. Overall, the first scenario with 30 robots and without disturbances is the simplest case we investigate, while the last scenario with 50 robots and disturbances constitutes the most complex case.

After having presented the experimental setup of the multi-robot observation scenario in this section, we will look in the next section 6.2 at the parameter settings used for the algorithms to optimise the scenarios shown in Fig. 6.5 together with the corresponding experimental results.

6.2 Experimental Results

6.2.1 Parameter Settings

We have performed an intensive parameter study to determine the optimal parameters for DE, PSO, SA, GA and RBI. In the following, we present the parameter settings used for the algorithms:

The Parameter Setting for DE

In case of DE, a large scaling factor F produces a very bad system performance in the multi-robot observation scenario so that the repulsion parameters P_i do not converge and remain distributed in the search space over the whole simulation time. The setting $F = 0.5$, which we have used to optimise the static functions presented in Sec. 5.1, provides such a result. Thus, we have decreased F in order to guarantee the convergence. Here, we have observed that the agents lose their exploration capabilities resulting in a premature convergence, if we use a very small value for F such as 0.1. In this context, the parameter combination $F = 0.275$ and $Cr = 0.9$ has produced the best results for DE.

The Parameter Setting for PSO

For PSO, we have compared the parameter settings proposed in [103] and in [105]. We have observed that the parameter setting proposed by Eberhart et al. in [103] produces a better system performance regarding the total number of observations. In this context, the inertia weight (ω) is set to 0.729, while the particle increment (φ_1) and the neighbourhood increment (φ_2) are both set to 1.49445. We have also tested different neighbourhood sizes and determined that PSO performs best, if we set the neighbourhood size to 5.

The Parameter Setting for GA

For GA, we set the probabilities of mutation (p_m) and crossover (p_c) to 0.1 and 0.9, respectively. In this context, we have observed that a mutation operator with an annealing scheme results in a premature convergence so that GA produces a bad system performance. Thus, the mutation is implemented using the Gaussian mutation operator without an annealing scheme, where the current solution of an individual is mutated with the standard deviation of 1. The crossover is implemented using the arithmetic crossover operator, where a tournament selection with a tournament size of two individuals is used to select individuals for crossover.

The Parameter Setting for SA

For SA, we use the same distribution function [131] that is used in function optimisation presented in Sec. 5.1. Here, we set the cooling ratio to 0.6 and the initial temperature to 10e+6.

The Parameter Setting for RBI

For RBI, we set α to 0.2, which corresponds to $1/100^{th}$ of the search space defined between [-10, 10], and β to 1.5. The neighbourhood size (i.e., the number of agents in the comparison set) is set to 60% of the population size (e.g., the neighbourhood size is 18 for the population that consists of 30 agents).

6.2.2 Convergence Speed of RBI

In this section, we discuss the convergence speed of RBI, DE, PSO, GA and SA in the multi-robot observation scenario. In this context, the system performance obtained by each algorithm depends on the convergence speed of the corresponding algorithm, since we use the total number of observations as the performance measure (see Eq. 6.1). Thus, the algorithm, which finds a particular solution in a shorter time produces a better system performance than another algorithm, which requires more time to find the same solution. We measure the convergence speed of the algorithms by determining the number of function evaluations (see Eq. 6.5) required to achieve a specific success criterion as in Sec. 5.1.4. We use Scenario 1 and Scenario 2 presented in Fig. 6.5 to determine the convergence speed of each algorithm. We count the number of function evaluations required by each algorithm to achieve 750 observations in Scenario 1 with 30 robots and 1500 observations in Scenario 2 with 50 robots. 20 experiments, each initialised with a different random seed, are carried out for each algorithm, and the average number of function evaluations are recorded. Fig. 6.6 shows the experimental results.

#FE	DE	PSO	RBI	GA	SA	Sucess criterion
Scenario 1 (30 robots)	3759.71	2731.99	1785.83	2258.11	5748.57 (*)	>750 observations
Scenario 2 (50 robots)	9257.31 (*)	6251.52	3481.84	4558.38	12519.17 (*)	>1500 observations

Figure 6.6: The convergence speed of RBI, DE, PSO, GA and SA measured in terms of the number of function evaluations. The success criteria for Scenario 1 and Scenario 2 are 750 and 1500 observations, respectively (see Eq. 6.1). Best results are shown in grey.

During our experiments in Scenario 1, we have observed that all algorithms except for SA converge to a specific range of solutions in the search space defined between -10 and 10, while the repulsion parameters P_i with SA remain distributed over the whole simulation time. However, SA also achieves the success criterion of 750 observations after a considerably long time, since we measure the system performance by determining the *sum* of all numbers of observations, and the robots, which optimise using SA, observe the target randomly from time to time without covering or cornering it. Thus, we mark

SA with (*) to denote that SA does not produce a specific solution and cannot create a collective group behaviour in this scenario. The results presented in Fig. 6.6 show that RBI has the highest convergence rate achieving the corresponding success criterion after 1785.83 function evaluations on average. It is interesting to compare GA and PSO. PSO is actually well-known for its high convergence speed in static fitness landscapes in the literature [23]. The situation looks different in self-referential fitness landscapes, where GA has a higher convergence speed than PSO resulting in 750 observations after 2258.11 function evaluations, while PSO requires 2731.99 evaluations on average to achieve the same result. Our experiments have shown that DE requires 3759.71 function evaluations on average to achieve 750 observations. We have observed that DE converges very slowly and needs a considerably longer time than RBI, PSO and GA to produce the same result.

The experiments in Scenario 2 have shown interesting results. We have observed that neither SA nor DE can cope with the increasing number of robots in the system. They don't converge to a specific solution (or range of solutions) in the search space defined between -10 and 10. The repulsion parameters with SA remain distributed over the whole simulation time so that SA shows again the lowest convergence speed (12519.17) within the set of investigated algorithms. DE produces an interesting behaviour in this more complex scenario. Here, we have observed two successively occurring situations when the robots optimise their behaviour with DE. These are the situations, (1) where the repulsion parameters are completely distributed in the search space [-10, 10] as in the case of SA, and (2) where the repulsion parameters *tend* to converge to a solution between -2 and 4. At the beginning of the optimisation, we observe the situation (1), where the repulsion parameters are distributed in the search space so that no specific solution can be identified. After some time, the repulsion values tend to converge to a specific range of solutions between -2 and 4 so that the individuals represents solutions that are only in this part of the search space ([-2,4]). At this point, DE cannot preserve this convergence process and the individuals accept solutions outside of the convergence range [-2,4] again so that the situation (1) re-occurs. Thus, it is not possible to identify a specific solution with DE in this more complex scenario. However, DE performs

better than SA due to the situation (2) stated above. We have marked DE and SA with (*) to denote that none of them provides a specific solution for the system with 50 robots. Regarding the other optimisation algorithms, RBI has the highest convergence speed requiring 3481.84 function evaluations on average for 1500 observations. The difference between the convergence speed of GA and PSO becomes more apparent in this scenario. Here, GA scales well with the increasing number of robots and provides 1500 observations after 4558.38 function evaluations, while PSO requires 6251.52 function evaluations on average to produce the same result. Overall, the results presented in this section give the clear proof that RBI has the highest convergence speed in both low-complex and high-complex scenarios with 30 and 50 robots.

After having investigated the convergence speed of the algorithms in this section, we investigate in Sec. 6.2.3 and in Sec. 6.2.4 the system performance obtained by each algorithm in scenarios without and with disturbances, respectively.

6.2.3 Observation Scenario without Disturbances

In this section, we present the experimental results for the system performance obtained by RBI, DE, PSO, GA and SA regarding the first two scenarios, Scenario 1 and Scenario 2, presented in Fig. 6.5. As mentioned before, these scenarios are used to compare the algorithms in an environment without disturbances. Scenario 2 is more complex than Scenario 1, since it involves more elements (robots), which influence the form of the fitness landscape (see Sec. 2.2). Here, we have carried out 20 experiments each with 50,000 iterations (ticks) for each scenario, and determined the average fitness value obtained by each algorithm. Fig. 6.7 shows the experimental results.

Our experiments show that in both scenarios (with 30 and 50 robots) RBI produces the best system performance. In Scenario 1 with 30 robots, the P_i values obtained by both DE and RBI converge to the same range of values between 1.2 and 1.6, which gives the best system performance for this scenario. The difference between DE and RBI is that DE requires approximately 30.000 ticks and RBI only 6.000-6.500 ticks for convergence. Thus, RBI's overall

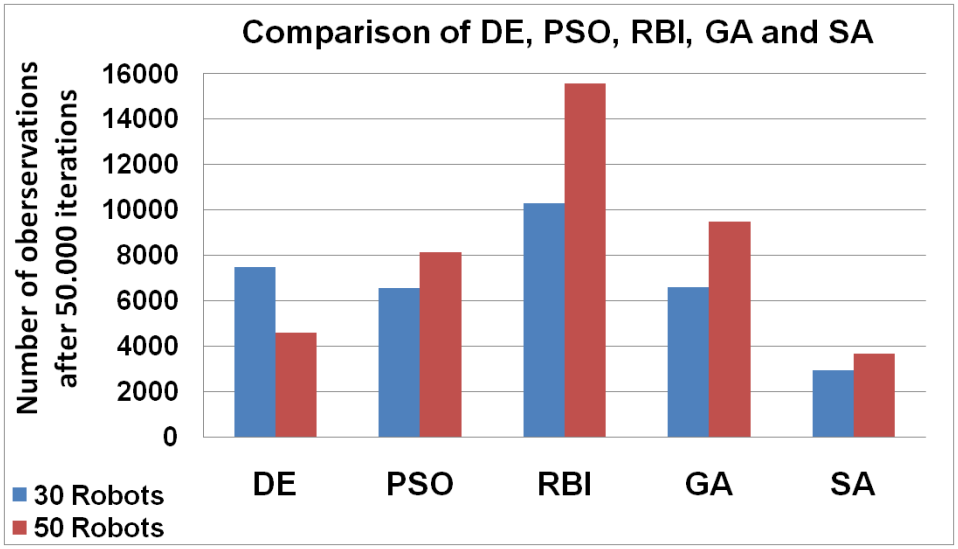


Figure 6.7: The total number of observations obtained by RBI, DE, GA, PSO and SA for the scenarios without disturbances.

performance is better than DE's in this scenario. PSO finds in this scenario only a suboptimal solution so that the P_i values converge to a range of values between 1.5 and 2.5. Although PSO gets stuck in a local optimum in [1.5, 2.5], its convergence speed is higher than DE (see Sec. 6.2.2) so that PSO compensates its weakness in finding accurate solutions with its convergence speed and produces a system performance that is comparable to the system performance obtained by DE. The system behaviour with GA is very similar to the system behaviour with PSO in this scenario. In this context, GA also gets stuck in a local optimum in [1.5, 2.5]. However, the convergence speed of GA is slightly higher than the converge speed of PSO so that GA achieves a total number of observations of 6612.95, while PSO provides 6545.15 observations on average in the same time. SA does not converge in this scenario so that the parameter values remain distributed in the search range [-10, 10] over the whole simulation time providing the worst system performance.

In the more complex scenario with 50 robots, small repulsion forces (i.e., small P_i values) lead to a better system performance, since we have a large number of robots in the same environment. Here, we have observed that the P_i values with RBI converge to a range of values between 0.4 and 0.8 providing

the best system performance. GA and PSO both get stuck in a local optimum in [0.5, 1.5], while GA scales better than PSO with the large number of robots in this scenario so that it has the higher convergence speed (see Sec. 6.2.2). Thus, the system performance obtained by GA is clearly better than the one obtained by PSO. In this scenario, DE cannot cope with the increasing number of robots so that no specific solution can be identified using DE. The P_i values with DE temporarily converge to the range of values between -2 and 4 so that the robots can only slightly increase the system performance. However, DE cannot preserve the convergence process as explained in Sec. 6.2.2 so that the P_i values become completely distributed in the search space after a specific amount of time. Thus, the robots optimising with DE can only slightly increase the system performance, and this explains why the system performance with DE in Scenario 1 is better than in Scenario 2. The P_i values with SA remain distributed in the search space [-10, 10] without any convergence over the whole simulation time so that it produces the worst system performance. Overall, only RBI, PSO and GA can cope with the increasing complexity and provide good results in this more complex scenario, while RBI outperforms all its competitors.

6.2.4 Observation Scenario with Disturbances

In this section, we present the experimental results regarding the last two scenarios, Scenario 3 and Scenario 4, presented in Fig. 6.5. These scenarios involve disturbances, and thus serve as a test-bed to determine to what extent the algorithms are capable of compensating the negative effects of disturbances. Here, we periodically set the repulsion parameters P_i of robots to a random value between -10 and +10 in order to create a disturbance. After each disturbance, the robots lose their learnt knowledge and the system performance decreases until they have adapted their repulsion parameters P_i to compensate the disturbance. In this context, we investigate two different levels of disturbances: (1) A low-level disturbance, where the disturbances occur every 10.000 ticks and (2) a high-level disturbance, where the disturbances occur every 5,000 ticks. In each case, we have carried out 20 experiments

each with 50,000 ticks and determined the average fitness value obtained by each algorithm. Fig. 6.8 shows the experimental results with 30 robots for both low-level and high-level disturbances. For comparison purposes, we have also included the results without disturbances obtained by the algorithms to determine the decrease in the system performance after disturbances.

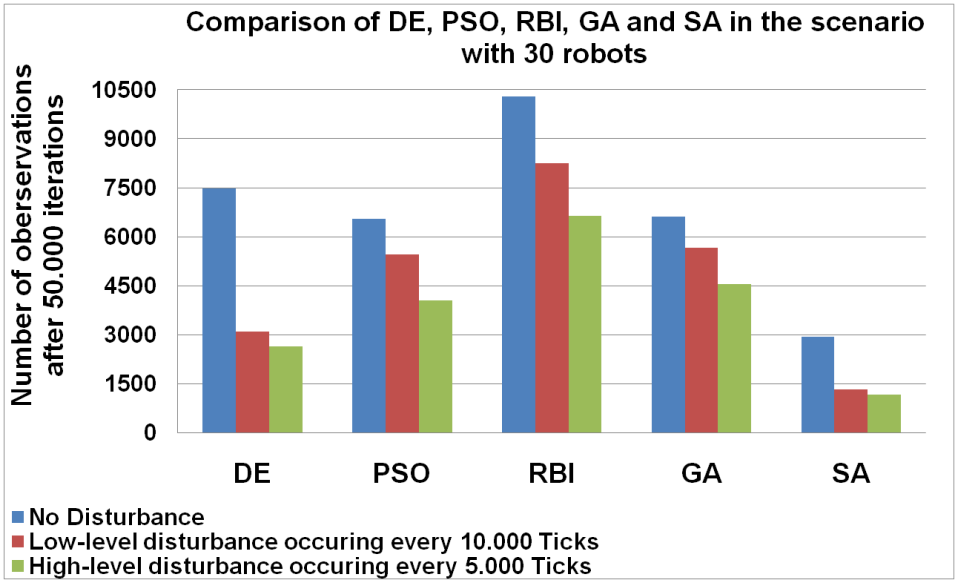


Figure 6.8: The total number of observations obtained by RBI, DE, GA, PSO and SA in Scenario 3, which consists of 30 robots and involves disturbances.

The results in Fig. 6.8 show that the system performance obtained by DE decreases drastically in the presence of disturbances. DE cannot cope with disturbances so that the system performance decreases more than 50% in both cases with low-level and high-level disturbances. The reason is that DE has a very low convergence speed (see Sec. 6.2.2) so that even the low-level disturbance occuring every 10.000 ticks can prevent it to find an acceptable solution until the next disturbance occurs. PSO and GA, on the other hand, can cope with disturbances in contrast to DE. The results show that GA is better than PSO in the investigated scenario with 30 robots confirming the results regarding the convergence speed presented in Fig. 6.6. On average, GA provides 189.6 obervations more than PSO in the scenario with low-level disturbances, while the difference is 515.3 observations in the scenario with high-level disturbances. Thus, GA seems to perform better than PSO in case

of high-level disturbances. RBI produces the best results in this scenario providing 8241.25 and 6628.85 observations in scenarios with low-level and high-level disturbances, respectively. The results show clearly that the effective optimisation scheme of RBI facilitates a very high convergence speed (6.2.2) in comparison to other algorithms so that the robots optimising with RBI can compansate the disturbances in a very small amount of time. For the sake of completeness, we have also investigated SA in this scenario considering the fact that SA does not converge even in the scenarios without disturbances as discussed in Sec. 6.2.3. As expected, the results show that SA is the worst performing algorithm within the set of investigated algorithms providing the lowest system performance.

The situation does not change significantly, if we increase the number of robots in the system. Fig. 6.9 shows the system performance obtained by the algorithms in the scenario that consists of 50 robots and involves disturbances.

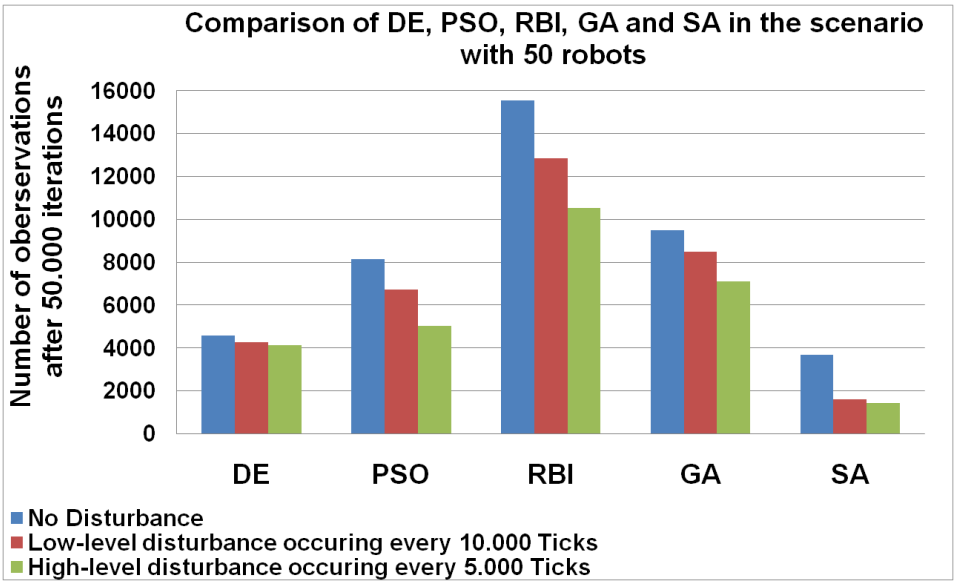


Figure 6.9: The total number of observations obtained by RBI, DE, GA, PSO and SA in Scenario 4, which consists of 50 robots and involves disturbances.

As discussed in Sec. 6.2.3, DE and SA do not converge and provide a specific solution in Scenario 2, which consists of 50 robots and does not involve disturbances. Thus, an additional increase of complexity using disturbances does not have a significant effect on the system performance obtained by DE

and SA. Hence, these algorithms produce still the worst results in this scenario, which consists of 50 robots and involves disturbances. If we look at the system performances achieved by PSO and GA, we see that GA outperforms PSO clearly providing 1766 and 2055.75 observations more than PSO in case of low-level and high-level disturbances, respectively. Hence, the results show that GA can find solutions that have at least the same quality as the solutions provided by PSO requiring less time than PSO. Thus, regarding the comparison of GA and PSO we can conclude that GA is the more convenient optimisation algorithm to be used in scenarios with high-level complexity. The results also show that RBI is the best performing algorithm in this most complex scenario giving the clear proof that (1) it can cope with low-level and high-level disturbances better than its competitors, and (2) it scales better than its competitors with the increasing number of robots, where each robot influences the form of the corresponding self-referential fitness landscape and makes the optimisation task harder.

An optimisation algorithm must find acceptable solutions in a small amount of time in order to maintain a reasonable system performance by adapting to changing environmental situations in case of disturbances. The standard notion for this issue is *robustness* [40, 10]. In this context, the results presented in Fig. 6.8 and in Fig. 6.9 show clearly that the system achieves a higher degree of robustness with RBI than with DE, PSO, GA and SA, since RBI provides high quality solutions in a short time. However, more investigations are required to gain a deeper understanding of robustness of a given system. Therefore, we investigate in the next section 6.3 a quantified definition of robustness according to a framework that is based on the *state space* modelling of the system behaviour as proposed in [40, 10]. Furthermore, we present experimental results regarding the degree of robustness achieved by DE, PSO, RBI, GA and SA.

6.3 Robustness in OC Systems

Robustness has different meanings depending on the context. Typical definitions include the ability of a system to maintain its functionality even in the

presence of changes in their internal structure or external environment [139], or the degree to which a system is insensitive to effects that have not been explicitly considered in the design [140]. In engineering, robust design generally means that the design is capable of functioning correctly, (or, at the very minimum, not failing completely) under a large range of conditions. It is also often related to manufacturing tolerances, and the corresponding literature is immense, see e. g., the works by Taguchi [141]. In scheduling, robustness of a plan generally means that it can be executed and will lead to satisfying results despite changes in the environment [142], while in computing, robustness is often associated with fault tolerance [143].

As there is a multitude of definitions of the term robustness, there are also many different possible metrics. For example, robustness can be judged by looking at the system performance over a distribution of scenarios, e. g., the average performance and its standard deviation, the signal-to-noise ratio, or the worst-case performance. Other metrics include the system's probability of failure, and the maximum deviation from standard conditions where the system can still cope with failures. A procedure for deriving a robustness metric for an arbitrary system is proposed by Shestak, Siegel et al. [144]. Robustness is defined in this context as a limited degradation of the system performance against perturbations with regard to specified system parameters. Waldschmidt used this procedure in [145] to derive a robustness definition for mixed signal systems. The robustness formula proposed there can be evaluated using simulations based on affine arithmetic [146].

Although there are many research projects done in measuring robustness in various domains, to our knowledge, a thorough investigation of robustness metrics for OC systems does not exist yet. Such metrics are needed to be able to design OC systems, which can limit the degradation of their performance against disturbances occurring inside or outside the system and adapt its behaviour accordingly.

In OC, we define robustness as the capability to adapt to (internal or external) changes and to maintain a required functionality in spite of a certain range of parameter variations. In this context, we formalise robustness using a framework based on the state space modelling of the system behaviour as

shown in Fig. 6.10 [40, 10].

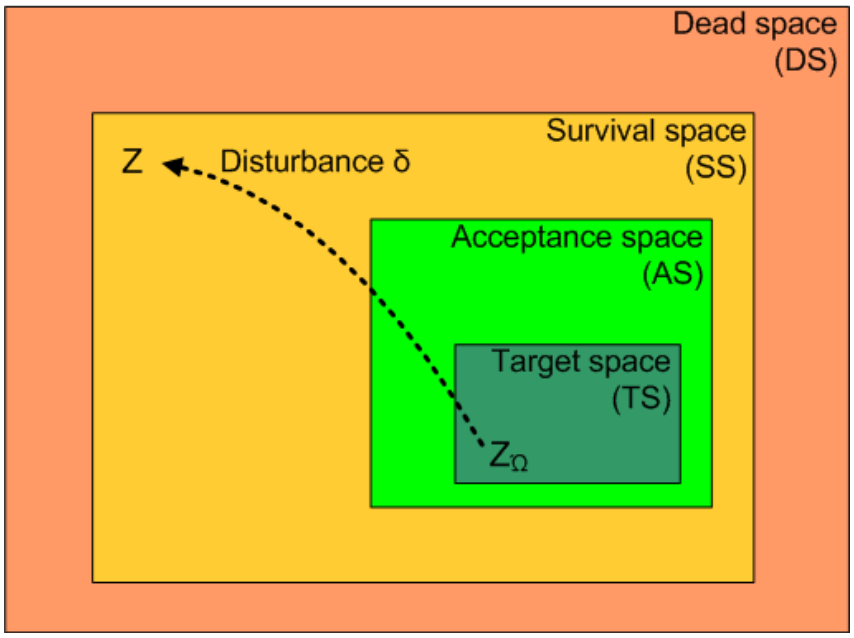


Figure 6.10: The state space of a system with several subspaces.

In the following, the terms given in Figure 6.10 are introduced briefly.

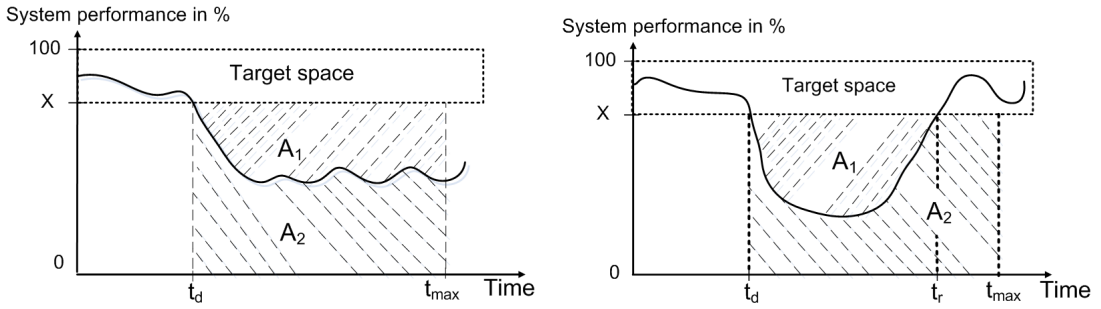
1. *Target space (TS)*: The target space consists of all states, where the system produces an optimal performance. All states in the target space are qualitatively equivalent regarding to the provided system performance.

2. *Acceptance space (AS)*: The acceptance space consists of states, where the system provides acceptable but not optimal performance. A system in the acceptance space can end up in the survival space, e.g., in case of a disturbance or it can move to another state in the target space after an *(self-) optimisation* process.

3. *Survival space (SS)*: The survival space consists of a maximal range of states including the acceptance space. A system, which is in the survival space but not in the acceptance space, is in a functional state providing highly degraded (possibly minimum) system performance. An OC system provides the corresponding interfaces for a control mechanism that can lead a system back into its acceptance space.

4. *Dead space (DS)*: A system leaving its survival space ends up in the dead space and is permanently damaged.

Based on this kind of modelling, we can determine the level of robustness monitored in the system according to the transitions between different subspaces. The occurrence of a *disturbance* would move the system, which ideally will be in its target space, to a state outside of its target space. In this context, the robustness can be characterised as the capability of a system to move again to an optimal state after the occurrence of a particular disturbance. Based on the framework presented Fig. 6.10, we investigate now two questions: (1) Is the system capable of moving back to its target space after the disturbance? and if so, (2) how much time does the system require to get back to a state in its target space? In this context, we consider two different cases as shown in Fig. 6.11.



(a) The system is not robust since it cannot return to its target space by t_{max}.

(b) The system is robust since it returns back to its target space at $t_r < t_{max}$.

Figure 6.11: The two cases, which may occur after a disturbance at t_d. The system is robust if it returns back to its target space within the recovery period defined by t_{max}.

In order to measure the robustness of a given system, we have to determine (1) a minimum system performance level (X) that the system must achieve to be in a state in the target space, and (2) a recovery period, which defines the maximum amount of time for the system to return back to its target space after the occurrence of a disturbance. In order to determine this recovery period, we consider t_d, which is the occurrence time of the disturbance, and t_{max}, which is the last point in time at which the system should be in its

target space again in order to show a robust behaviour. In this context, the system shown in Fig. 6.11(a) is not robust, since it cannot regain the minimum required system performance X within the recovery period $(t_{max} - t_d)$ after the occurrence of a disturbance at t_d. Fig. 6.11(b), on the other hand, shows a robust system behaviour, where the system returns back to its target space at t_r (t_r stands for the time of recovery) within the recovery period $(t_{max} - t_d)$. Here, the disturbance results in a decreased system performance, which can be determined using the area A_2, while the loss in the system performance corresponds to the area A_1 as presented in Fig. 6.11(b). In this context, we measure the robustness R of a given system S according to X, t_r, t_d and t_{max} using Eq. 6.6:

$$R_S(X, t_d, t_r, t_{max}) = \frac{A_2}{A_1 + A_2} = \frac{\int_{t_d}^{t_r} P_S(t)dt + X * (t_{max} - t_r)}{X * (t_{max} - t_d)} \tag{6.6}$$

where A_2 represents the actual system performance that is provided between t_d and t_{max}, while (A_1+A_2) represents the minimum system performance that should have been achieved to be always in the target space between t_d and t_{max}. Here, we determine A_2 considering (1) the decreased system performance between t_d and t_r ($\int_{t_d}^{t_r} P_S(t)dt$) and (2) the minimum required system performance X between t_r and t_{max}. In order to calculate $(A_1 + A_2)$, we simply multiply the minimum required system performance X with the time elapsed between t_{max} and t_d. The robustness is then determined by calculating $A_2/(A_1 + A_2)$. Please notice that $P_S(t)$ given in Eq. 6.6 is not a simple function that we can use to calculate the system performance after the disturbance, rather it is an observation of the system performance between t_r and t_d. Thus, in order to calculate $\int_{t_d}^{t_r} P_S(t)dt$, we observe and measure the system performance after the occurrence of the disturbance at time t_d and determine the overall decreased system performance until the time of recovery t_r is reached. Hence, the robustness of a given OC system cannot be calculated prior to the accomplishment of this observation and measurement process.

In the following, we measure the robustness of the system obtained by DE, PSO, RBI, GA and SA. In this context, we investigate two scenarios with 30

and 50 robots, where each scenario is simulated for 50,000 ticks. In order to investigate how the system reacts on the particular disturbance and how fast it can recover from this disturbance, we consider the number of observations provided every 500 ticks (sampling period) instead of calculating the sum of all number of observations provided until the end of the simulation. Thus, we have 100 consecutive samples (50,000 / 500) that we consider in our experiments. We use the same disturbance as presented in Sec. 6.2.4, where the repulsion parameters P_i of the robots are set to a random value between -10 and +10. In this context, we implement the disturbance so that it does not occur periodically, but only once at tick 25,000 (after 50 sampling periods), and give the system 5000 ticks (10 sampling periods) to recover from the disturbance and return back to its target space. Thus, in terms of sampling periods t_d is set to 50 and t_{max} to 60 corresponding to 25,000 and 30.000 ticks, respectively. This results in a recovery period that consists of 10 sampling periods ($t_{max} - t_d$). The parameter X, which defines the minimum system performance level that must be achieved to be in a state in the target space, is set to 60 observations per sampling period for the scenario with 30 robots and 90 observations per sampling period for the scenario with 50 robots. Fig. 6.12 and Fig. 6.14 show the system performance after the disturbance with 30 and 50 robots, respectively.

The results presented in Fig. 6.12 show that the system performance with DE, PSO, RBI, GA and SA decreases after the occurrence of the disturbance, while all algorithms except for SA recover from the disturbance after some time and provide 60 observations per sampling period. Although DE can compensate the disturbance, it needs clearly more time than the recovery period (10 sampling periods) to provide the minimum required system performance of 60 observations per sampling period. Thus, only PSO, RBI, and GA create a robust system behaviour according to the results presented in Fig. 6.12. In order the determine the robustness of the system provided by the algorithms, we calculate the area $A_1 + A_2$ as $(t_{max} - t_d) * X = (60 - 50) * 60 = 600$, while A_2 is calculated using $(t_{max} - t_r) * X$ and the system performance obtained between t_r and t_d as shown in Fig.6.11(b). The robustness obtained by each algorithm is presented in Fig. 6.13.

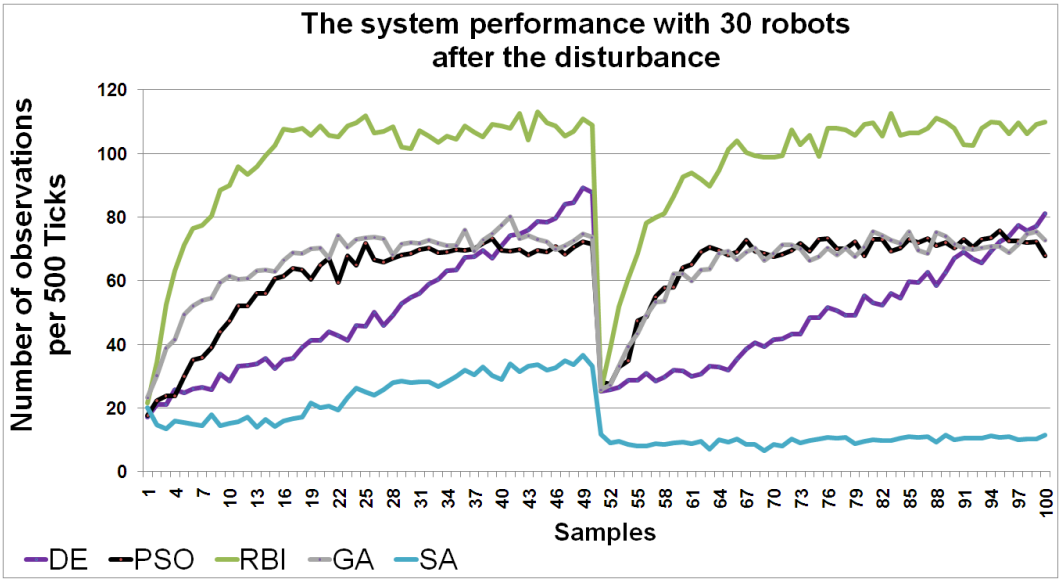


Figure 6.12: The system performance obtained by DE, PSO, RBI, GA and SA. In this scenario, the system consists of 30 robots, the minimum performance level X is set to 60 observations per sampling period (500 ticks), the disturbance occurs after 50 sampling periods (25,000 ticks) and the maximum amount of time for the system to return to its target space elapses 10 sampling periods (5,000 ticks) after the occurrence of the disturbance.

	DE	PSO	RBI	GA	SA
Disturbance time (td)	50	50	50	50	50
tmax	60	60	60	60	60
Recovery time (tr)	87	60	54	59	∞
A1 + A2 = (tmax - td) * X	600	600	600	600	600
A2	287	450	536	444	91
Robustness = A2 / (A1 + A2)	0	75%	89.33%	74%	0

Figure 6.13: The robustness obtained by DE, PSO, RBI, GA and SA in the scenario with 30 robots, where X is set to 60 observations per sampling period and recovery period ($t_{max} - t_d$) is limited to 10 sampling periods. t_{max}, t_r and t_d are given in terms of sampling periods (see Fig. 6.11 for the definition of t_{max}, t_r, t_d, A_1 and A_2).

The system with SA is not robust, since it cannot optimise the repulsion parameters P_i so that the system provides 60 observations per sampling period (X). DE requires 37 sampling periods (18.500 ticks) to achieve the intended

system performance X. Thus, neither DE nor SA can compensate the disturbance within the recovery period so that the robustness of the system is 0 in each case. The results show that GA and PSO provide almost the same (with 1% difference) robustness level, while the system achieves the highest level of robustness with RBI. RBI can recover from the disturbance faster than GA and PSO requiring only 4 sampling periods ($t_r = 54$) after the occurrence of the disturbance ($t_d = 50$), while GA and PSO require 10 ($t_r = 60$) and 9 ($t_r = 59$) sampling periods, respectively to achieve the same performance. Overall, the degree of robustness achieved with RBI is 89.33%, while PSO and GA achieve 75% and 74%, respectively. Fig. 6.14 presents the number of observations provided by the algorithms in every sampling period (500 ticks) for the system 50 robots.

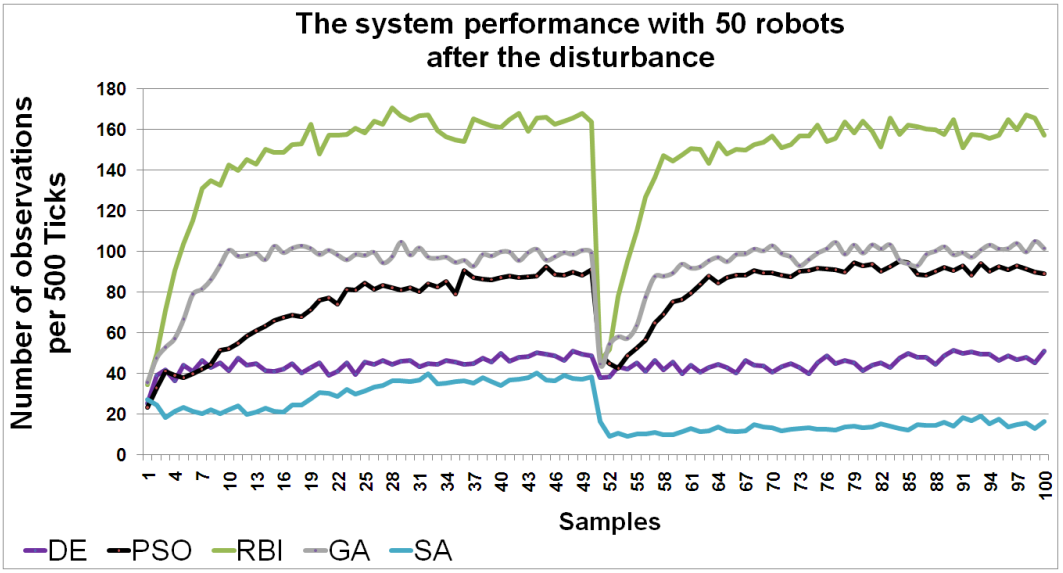


Figure 6.14: The system performance obtained by DE, PSO, RBI, GA and SA. In this scenario, the system consists of 50 robots, the minimum performance level X is set to 90 observations per sampling period (500 ticks), the disturbance occurs after 50 sampling periods (25,000 ticks) and the maximum amount of time for the system to return to its target space elapses 10 sampling periods (5,000 ticks) after the occurrence of the disturbance.

The results presented in Fig. 6.14 show that neither DE nor SA can cope with the complexity caused by the increased number of robots and recover from

the disturbance providing approximately 45 and 20 observations per sampling period, respectively. RBI, PSO and GA, on the other hand, compensate the disturbance after some time and provide the minimum required system performance for the target space, which is 90 observations per sampling period. We have observed that RBI requires considerably less time in comparison to PSO and GA to recover from the disturbance, while GA provides a better system performance than PSO in this more complex scenario. In this context, RBI recovers from the disturbance only 4 sampling periods after the disturbance ($t_r = 54$), while GA and PSO do the same requiring 10 ($t_r = 60$) and 13 ($t_r = 73$) sampling periods, respectively. Since the recovery period is set to 10 sampling periods (5,000 ticks), only RBI and GA achieve a robust system behaviour providing the minimum required system performance X within the recovery period. Although PSO can also recover from the disturbance, it does not provide the intended system performance until t_{max} so that the system behaviour with PSO is categorised as "not robust" as well as the system behaviour with DE or SA. The robustness obtained by each algorithm is presented in Fig. 6.15.

	DE	PSO	RBI	GA	SA
Disturbance time (td)	50	50	50	50	50
tmax	60	60	60	60	60
Recovery time (tr)	∞	73	54	60	∞
A1 + A2 = (tmax - td) * *X*	900	900	900	900	900
A2	420	578	805	709	106
Robustness = A2 / (A1 + A2)	0	0	89.44%	78.77%	0

Figure 6.15: The robustness obtained by DE, PSO, RBI, GA and SA in the scenario with 50 robots, where X is set to 90 observations per sampling period and recovery period ($t_{max} - t_d$) is limited to 10 sampling periods. t_{max}, t_r and t_d are given in terms of sampling periods (see Fig. 6.11 for the definition of t_{max}, t_r, t_d, A_1 and A_2).

As mentioned above, PSO, DE and SA cannot provide a robust system behaviour based on their recovery times (t_r). The recovery times of DE and SA are both ∞ and the recovery time of PSO is 73 so that in each case we have $t_r > t_{max}$ providing a robustness of 0%. RBI recovers 4 sampling periods after the

disturbance ($t_r = 54$) and provides 805 oberservations within this 4 sampling periods, while GA recovers 10 sampling periods after the disturbance ($t_r = 60$) and provides 709 obersations within this 10 sampling periods. Accordingly, the robustness achived by RBI 89.44% and the robustness achieved by GA is 78.77%. Thus, RBI outperforms all its competitors in this more complex scenario with 50 robots providing the highest level robustness for the system. Overall, the results presented in Fig. 6.13 and in Fig. 6.15 give the clear proof that RBI is the most effecient algorithm within the set of investigated algorithms achieving the highest level of robustness for the system with the self-referential fitness landscape presented in this section.

6.4 Summary

In this section, we have investigated the performance of DE, PSO, RBI, GA and SA in the multi-robot observation scenario from the predator/prey domain. The presented scenario defines a problem with a self-referential fitness landscape, where the solution-fitness mapping changes as a function of agent (robot) behaviour. In this context, the fitness value of a particular repulsion parameter P_i changes based on the distribution of all P_i values in the system resulting in a highly dynamic fitness landscape that changes each time the robots optimise their P_i values. Hence, in order to optimise in such a fitness landscape the corresponding optimisation algorithm must cope with this complexity guaranteeing (1) the convergence of solutions and (2) the exploration of new solutions without forgetting the already found good solutions. In this context, we have investigated 4 different observation scenarios (see Fig. 6.5) with an increasing level of complexity and compared the optimisation algorithms DE, PSO, RBI, GA and SA with each other. The experimental results have shown that RBI can effectively cope with the increasing level of complexity in the presented self-referential fitness landscape providing the highest quality solutions in a short period of time. The results are summarised in Fig. 6.16.

The results have also shown that GA scales well with the increasing number of robots and performs better than PSO especially in scenarios with a higher

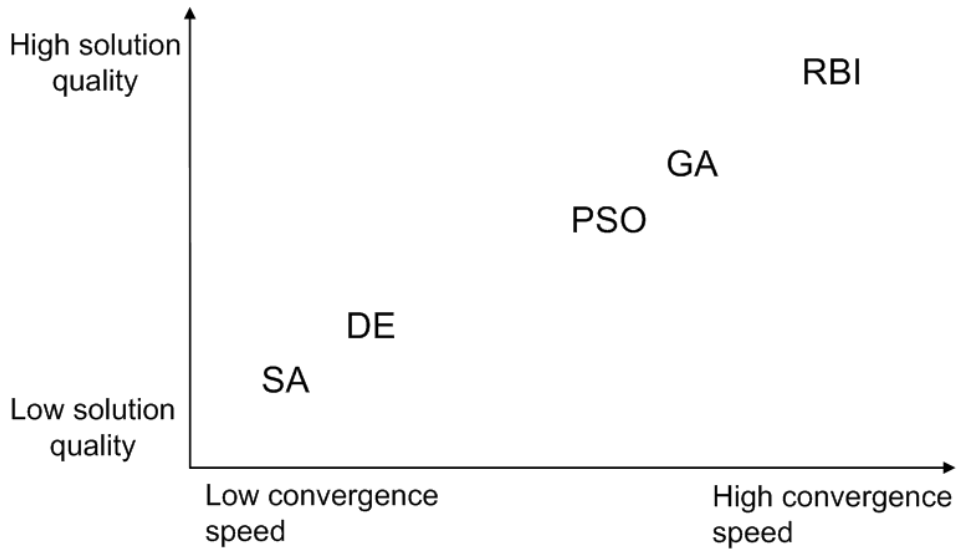


Figure 6.16: The comparison of convergence speeds and the quality of solutions provided by RBI, DE, PSO, GA and SA in the multi-robot observation scenario.

level of complexity, while PSO outperforms DE in the corresponding scenarios (see scenario 2-3-4 in Fig. 6.5). DE provides good results only in Scenario 1, which is the simplest scenario with 30 robots and without disturbances, while it cannot cope with the increasing level of complexity investigated in Scenarios 2, 3 and 4. Thus, the results obtained by DE are worse than the results obtained by RBI, PSO and GA. In none of the investigated scenarios, the robots optimising with SA can cover the target and increase the number of observations effectively so that SA provides the worst results in all of the investigated scenarios.

After having investigated the convergence speed and system performance obtained by the algorithms, we presented a framework based on the state space modelling of the system behaviour to provide a quantified notion of robustness. In this context, we have elaborated the effects of the convergence speed and solution quality achieved by the algorithms on the robustness of the system. Our experiments have shown that RBI is the most effective optimisation algorithm within the set of investigated algorithms, and can recover from the disturbances in a very small amount of time using its highly efficient exploration/exploitation scheme. Overall, all the aspects investigated in this

chapter such as convergence speed, solution quality and robustness suggest the use of RBI to optimise in self-referential fitness landscapes, since it provides the best performance regarding all these aspects.

Chapter 7

Conclusion and Outlook

"The saddest summary of a life contains three descriptions:
could have, might have, and should have."
Louis E. Boone

7.1 Summary

Motivated by OC, the goal of this thesis has been to investigate and identify the challenges of optimisation in OC fitness landscapes, and develop an suitable optimisation algorithm, which finds high-quality solutions in a short time, to allow an OC system to react quickly to the changes in its environment. In this context, we have provided a classification of OC fitness landscapes based on the solution-fitness mapping used by the particular optimisation algorithm to determine the optimum. We have distinguished between the static and dynamic fitness landscapes. The main difference between the static and the dynamic fitness landscapes is that the solution-fitness mapping of a static fitness landscape does not change over time, while the solution-fitness mapping of a dynamic one changes e.g., as a function of agent behaviour. We have identified this type of fitness landscapes as *self-referential*. The agents (or individuals in terms of EA), which optimise in a self-referential fitness landscape, influence directly the solution-fitness mapping so that the fitness of a single

agent does not only depend on its own behaviour, but also on the behaviour of other agents in the system. Based on this classification, we have determined the following requirements that an optimisation algorithm must satisfy to be used in an OC system. The corresponding optimisation algorithm should provide:

1. a consistent exploration of the fitness landscape over the whole optimisation process to be able to track the moving optimum, and

2. a clear distinction between the exploring and exploiting individuals to minimise the time required to determine the new optimum after the form of the fitness landscape has changed.

We have developed a novel optimisation algorithm, the Role-based Imitation algorithm (RBI), which satisfies these requirement, and compared it to different state-of-the-art population-based and trajectory-based optimisation algorithms from the literature. For comparison purposes, we have considered different static and self-referential fitness landscapes, and measured the convergence speed and the quality of solutions obtained by the algorithms as comparison criteria. At this point, we complete the evaluation strategy presented in Sec. 4.4 with the corresponding results provided in Chapter 5 and in Chapter 6 as shown in Fig. 7.1.

To study the performance of RBI in problems defined over continuous variables with static fitness landscapes, we have implemented 21 different benchmark functions from the literature. The investigation of noiseless and noisy functions presented in Sec. 5.1 have shown that RBI outperforms DE, PSO, GA and SA regarding the convergence speed and the quality of solutions obtained by the algorithms. Particularly, the results regarding the noisy functions give the clear proof that the distinction of exploring and exploiting agents based on our novel role assignment strategy provides the whole population with the capability to cope effectively with the existing noise in the environment.

In order to evaluate the performance of RBI in dynamic and self-referential fitness landscapes, we have implemented the multi-robot observation scenario from the predator/prey domain. The decision on the choice of a scenario

	DE	PSO	GA	SA	ACO	RBI
Applicable in						
continous search spaces	Yes	Yes	Yes	Yes	No	Yes
discrete search spaces	No	No	Yes	Yes	Yes	Yes
Continous search spaces						
Static fitness landscapes						
Solution quality in noiseless env.	++	+	-	--	NA	++
Solution quality in noisy env.	-	+	-	--	NA	++
Convergence speed	-	+	-	--	NA	++
Self-referential fitness landscapes						
Solution quality in noiseless env.	--	-	+	--	NA	++
Solution quality in noisy env.	--	-	+	--	NA	++
Convergence speed	--	-	+	--	NA	++
Discrete search spaces: TSP						
Static fitness landscapes						
Solution quality in noiseless env.	NA	NA	+	--	++	++
Convergence speed	NA	NA	-	--	++	+

Figure 7.1: The comparison of DE, PSO, GA, SA, ACO and RBI according to the evaluation strategy presented in Sec. 4.4. NA stands for "Not Available", while the symbols "+" and "-" indicate the higher and the lower performance, respectively.

from the predator/prey domain was based on fact that scenarios from this domain present a *generic* model/abstraction for many multi-agent systems (MAS), and especially for robotic MAS, since they involve agents, which move around in an environment and interact with each other in order to *collectively* accomplish a given task. In this context, we have compared RBI to DE, PSO, GA and SA. As comparison criteria, we have used the convergence speed and the quality of solutions obtained by the algorithms. Our experiments have shown that RBI can successfully provide (1) a consistent exploration of the fitness landscape over the whole optimisation process that is required to track the moving optimum, and (2) a clear distinction between the exploring and exploiting individuals to minimise the time required to determine the new optimum after the form of the fitness landscape has changed. In this context, RBI has outperformed all its competitors, and found highest quality of solutions in the shortest time. Furthermore, we have presented in Sec. 6.3

a framework based on the state space modelling of the system behaviour to provide a quantified notion of robustness. There, we have investigated the level of robustness achieved by RBI, DE, PSO, GA and SA in the multi-robot observation scenario. Our experiments have shown that RBI is the most effective optimisation algorithm within the set of investigated algorithms, and can recover from the disturbances in a very small amount of time using its highly efficient exploration/exploitation scheme providing the highest level of robustness (see Chapter. 6).

We have investigated the performance of RBI in search spaces defined over discrete variables using different instances of the well-known Traveling Salesman Problem (TSP). In this context, we have compared RBI to SA, GA and ACO. The experiments have shown that RBI and ACO outperform GA and SA, while the results obtained by ACO are slightly better than the ones obtained by RBI. Furthermore, ACO has found its solutions in a smaller amount of time in comparison to other algorithms so that it has the highest convergence speed within the set of the investigated algorithms. At this point, it is important to mention that there are two main restrictions on the set of problems that can be optimised with ACO. The first restriction is that ACO can only be used to solve problems, which have a graph representation, since it is particularly developed to find the shortest path in a given graph using the environment as a communication medium (stigmergy). On the other hand, RBI is applicable to all kind of discrete and continuous optimisation problems. The second restriction arises in case of optimisation in self-referential fitness landscapes. There, ACO would face difficulties in terms of convergence, since the ants need to increase the pheromone density on the shortest path in order to converge, and this "shortest path" changes continuously in a self-referential fitness landscape based on the exploration and exploitation behaviour of the ants. In contrast, RBI follows another optimisation pattern, which guarantees the convergence of solutions, where the agents find a (possibly suboptimal) solution at the beginning of the optimisation and improve this solution step by step in the course of the optimisation. Although ACO outperforms RBI, we have shown that the dynamic role assignment strategy of RBI can successfully be applied to problems defined over discrete variables. Thus, our results

provides a valuable reference for the further improvement of RBI for discrete search spaces.

Overall, our experiments have shown that RBI is a fast optimisation algorithm that produces high quality solutions for problems defined over continuous and discrete variables. One further advantage of RBI is the time required to configure RBI for a specific problem (or class of problems). Since RBI uses a small number of parameters (see Sec. 4.2), it can be quickly configured for the considered set of problems, which is a particular challenge for other optimisation algorithms like PSO (see Sec. 3.3.2).

7.2 Future Research Opportunities

Based on the presented experimental results, the main future research opportunity arises towards the *parallelisation* of RBI. In this context, the parallel version of RBI can be investigated in order to reduce the optimisation time for solving large-scale problems. Generally, population-based optimisation algorithms are suitable candidates for an efficient parallelisation, where the individuals (or agents) of a particular algorithm can run on different processors of a PC-cluster [147, 148]. This opens new possibilities for determining the adequate balance between the exploiting and exploring individuals to speed up the optimisation process and to improve the solution quality. In the following, we discuss some of these possibilities in more detail.

Partitioning of the Search Space

In this thesis, we have investigated a single population of RBI agents that try to determine the optimum in a single search space. In order to increase the performance of the algorithms, it is possible to partition the search space into distinct sectors and implement multiple populations of RBI agents, where each of them optimises in a particular sector of the search space. Here, the populations can communicate with each other in order to exchange information about the solutions they have found in their corresponding sectors, and try to collectively determine the global optimum. In this context, we can make

use of a parallel computer architecture to implement each population on a different processor (or group of processors) and to realise the communication (1) between the agents inside of each population and, (2) between the populations. Here, different problems may arise regarding the local synchronisation between the agents of a population and the global synchronisation between the populations. Thus, these problems should be solved first to exploit the advantage of the use of multiple populations.

Using Hybrid Algorithms

On parallel computer architectures, it is also possible to combine different algorithms in order to solve large-scale optimisation problems [149]. Basically, population-based optimisation algorithms, such as RBI and EA, have good exploration capabilities, while the local search (LS) algorithms, such as Stochastic Hill Climbing, are good candidates to refine the best solutions found so far. In this context, it is possible to make use of a parallel computer architecture in order to combine RBI with a particular LS algorithm. Here, RBI can be used to quickly identify the promising areas in the search space facilitating a coarse-grained optimisation, while an adequate LS algorithm can be used concurrently to refine the best solutions found so far facilitating a fine-grained optimisation.

In this thesis, we have achieved to answer only a few questions that arose regarding the optimisation in OC systems. Thus, the two research opportunities presented above do not cover all possible issues, which can be investigated in the future. Rather, they only show the direction for future work, which includes the ideas presented in this dissertation. The use of multiple populations and hybrid algorithms raises some new questions regarding the cooperation and coordination of different entities (i.e., individuals, populations etc.), which requires the consideration of different research areas such as Organic Computing, Multi-Agent Systems, Parallel Computing and numerical optimisation at the same time.

Bibliography

[1] Shu-Heng Chen, *Evolutionary Computation in Economics and Finance*, Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2002.

[2] Mitchell Potter and Kenneth De Jong, "A cooperative coevolutionary approach to function optimization," in *Parallel Problem Solving from Nature - PPSN III*, Yuval Davidor, Hans-Paul Schwefel, and Reinhard Männer, Eds., vol. 866 of *Lecture Notes in Computer Science*, pp. 249–257. Springer Berlin / Heidelberg, 1994.

[3] Hongmei Yan, Yingtao Jiang, Jun Zheng, Chenglin Peng, and Shouzhong Xiao, "Discovering critical diagnostic features for heart diseases with a hybrid genetic algorithm.," in *METMBS'03*, 2003, pp. 406–409.

[4] R. L. Johnston, *Applications of Evolutionary Computation in Chemistry*, Springer, Berlin, Germany, 2004.

[5] Charles Darwin, *On the origin of species*, New York :D. Appleton and Co.,, 1871, http://www.biodiversitylibrary.org/bibliography/28875.

[6] Thomas Weise, *Global Optimization Algorithms – Theory and Application*, it-weise.de (self-published): Germany, 2009.

[7] Organic Computing, "Website," `http://en.wikipedia.org/wiki/Organic_computing`.

[8] Christian Muller-Schloer, "Organic computing - on the feasibility of controlled emergence," in *CODES+ISSS '04: Proceedings of the international conference on Hardware/Software Codesign and System Synthesis*, Washington, DC, USA, 2004, pp. 2–5, IEEE Computer Society.

[9] Hartmut Schmeck, "Organic computing - a new vision for distributed embedded systems," in *ISORC '05: Proc. of the Eighth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing*, Washington, DC, USA, 2005, pp. 201–203, IEEE Computer Society.

[10] Hartmut Schmeck, Christian Müller-Schloer, Emre Cakar, Moez Mnif, and Urban Richter, "Adaptivity and self-organization in organic computing systems," *ACM Transactions on Autonomous and Adaptive Systems*, vol. 5, pp. 10:1–10:32, September 2010.

[11] Sven Tomforde, Holger Prothmann, Jürgen Branke, Jörg Hähner, Moez Mnif, Christian Müller-Schloer, Urban Richter, and Hartmut Schmeck, "Observation and control of organic systems," in *Organic Computing - A Paradigm Shift for Complex Systems*, Christian Müller-Schloer, Hartmut Schmeck, and Theo Ungerer, Eds., incollection 4.1, pp. 325–338. Birkhäuser, Juni 2011.

[12] Fabian Rochner, Holger Prothmann, Jürgen Branke, Christian Müller-Schloer, and Hartmut Schmeck, "An organic architecture for traffic light controllers," in *Informatik 2006 – Informatik für Menschen*, Christian Hochberger and Rüdiger Liskowsky, Eds. Oktober 2006, vol. P-93 of *Lecture Notes in Informatics (LNI)*, pp. 120–127, Köllen Verlag.

[13] Stewart W. Wilson, "Generalization in the XCS classifier system," *Genetic Programming 1998: Proceedings of the Third Annual Conference*, pp. 665–674, 1998.

[14] S. Kirkpatrick, C. D. Gelatt, Jr., and M. P. Vecchi, "Optimization by simulated annealing," *Science*, vol. 220, pp. 671–680, 1983.

[15] T. Back, U. Hammel, and H. P. Schwefel, "Evolutionary computation: comments on the history and current state," *Evolutionary Computation, IEEE Transactions on*, vol. 1, no. 1, pp. 3–17, April 1997.

[16] Sewall Wright, "The roles of mutation, inbreeding, crossbreeding, and selection in evolution," *Proceedings of the Sixth International Congress on Genetics*, 1932.

[17] S. Gavrilets, "Fitness landscapes and the origin of species," *Austral Ecology*, vol. 30, no. 5, pp. 610–611, 2004.

[18] Zbigniew Michalewicz and David B. Fogel, *How to Solve It: Modern Heuristics*, Springer, December 2004.

[19] Colin R. Reeves, "Fitness landscapes and evolutionary algorithms," in *AE '99: Selected Papers from the 4th European Conference on Artificial Evolution*, London, UK, 2000, pp. 3–20, Springer-Verlag.

[20] Stephen D. Turner, Marylyn D. Ritchie, and William S. Bush, "Conquering the needle-in-a-haystack: How correlated input variables beneficially alter the fitness landscape for neural networks," in *EvoBIO '09: Proceedings of the 7th European Conference on Evolutionary Computation, Machine Learning and Data Mining in Bioinformatics*, 2009, pp. 80–91.

[21] Jurgen Branke, *Evolutionary Optimization in Dynamic Environments*, Kluwer Academic Publishers, Norwell, MA, USA, 2001.

[22] Jeffrey Horn and David E. Goldberg, "Genetic algorithm difficulty and the modality of fitness landscapes," in *Foundations of Genetic Algorithms 3*. 1994, pp. 243–269, Morgan Kaufmann.

[23] Jakob S. Vesterstrøm and René Thomsen, "A comparative study of differential evolution, particle swarm optimization, and evolutionary algorithms on numerical benchmark problems.," in *Proceedings of the 2004 Congress on Evolutionary Computation*, 2004, vol. 2, pp. 1980–1987.

[24] Holger Prothmann, Fabian Rochner, Sven Tomforde, Jürgen Branke, Christian Müller-Schloer, and Hartmut Schmeck, "Organic control of traffic lights," in *Proceedings of the 5th International Conference on Autonomic and Trusted Computing (ATC-08)*. Juni 2008, vol. 5060 of *LNCS*, pp. 219–233, Springer.

[25] Emre Cakar and Christian Müller-Schloer, "Self-organising interaction patterns of homogeneous and heterogeneous multiagent populations," in

Proceedings of the 3rd IEEE International Conference on Self-Adaptive and Self-Organizing Systems, 2009, pp. 165–174.

[26] Emre Cakar, Sven Tomforde, and Christian Müller-Schloer, "A role-based imitation algorithm for the optimisation in dynamic fitness landscapes," in *IEEE Swarm Intelligence Symposium, 2011. SIS 2011*, Paris, France, 2011, pp. 139–146.

[27] Emre Cakar, Nugroho Fredivianus, Jörg Hähner, Jürgen Branke, Christian Müller-Schloer, and Hartmut Schmeck, "Aspects of learning in oc systems," in *Organic Computing - A Paradigm Shift for Complex Systems*, Christian Müller-Schloer, Hartmut Schmeck, and Theo Ungerer, Eds., incollection 3.1, pp. 237–251. Birkhäuser, Juni 2011.

[28] D. Challet and Y Zhang, "Emergence of cooperation and organization in an evolutionary game," *Physica A*, vol. 226, pp. 407–418, 1997.

[29] Arlindo Silva, Ana Neves, and Ernesto Costa, "An empirical comparison of particle swarm and predator prey optimisation," in *AICS '02: Proceedings of the 13th Irish International Conference on Artificial Intelligence and Cognitive Science*, London, UK, 2002, pp. 103–110, Springer-Verlag.

[30] Marco Dorigo and Thomas Stützle, *Ant Colony Optimization*, Bradford Company, Scituate, MA, USA, 2004.

[31] R. Takahashi, "Solving the traveling salesman problem through genetic algorithms with changing crossover operators," in *Machine Learning and Applications, 2005. Proceedings. Fourth International Conference on*, December 2005, p. 6 pp.

[32] Christian Höhn and Colin Reeves, "The crossover landscape for the onemax problem," in *Proceedings of the 2nd Nordic Workshop on Genetic Algorithms*, 1996, pp. 27–43.

[33] Sven Tomforde, Holger Prothmann, Fabian Rochner, Jürgen Branke, Jörg Hähner, Christian Müller-Schloer, and Hartmut Schmeck, "De-

centralised progressive signal systems for organic traffic control," in *Proceedings of the 2nd IEEE International Conference on Self-Adaption and Self-Organization (SASO 2008)*, Sven Brueckner, Paul Robertson, and Umesh Bellur, Eds. Oktober 2008, pp. 413–422, IEEE.

[34] Kalyanmoy Deb, J. Sundar, Udaya Bhaskara Rao N, and Shamik Chaudhuri, "Reference point based multi-objective optimization using evolutionary algorithms," in *International Journal of Computational Intelligence Research*. 2006, pp. 635–642, Springer-Verlag.

[35] C. A. C. Coello, "An updated survey of evolutionary multiobjective optimization techniques: state of the art and future trends," in *Evolutionary Computation, 1999. CEC 99. Proceedings of the 1999 Congress on*, 1999, vol. 1, pp. 3 vol. (xxxvii+2348).

[36] Michael Woolridge and Michael J. Wooldridge, *Introduction to Multiagent Systems*, John Wiley & Sons, Inc., New York, NY, USA, 2009.

[37] Kurt Konolige, Charles Ortiz, Regis Vincent, Andrew Agno, Michael Eriksen, Benson Limketkai, Mark Lewis, Linda Briesemeister, Enrique Ruspini, Dieter Fox an Jonathan Ko, Benjamin Stewart, and Leonidas Guibas, "CentiBOTS: Large scale robot teams," in *In AAMAS*, 2003.

[38] Roderich Groß, Michael Bonani, Francesco Mondada, , and Marco Dorigo, "Autonomous self-assembly in a swarm-bot," in *Proceedings of the 3rd International Symposium on Autonomous Minirobots for Research and Edutainment (AMiRE 2005)*, Kazuyuki Murase, Kosuke Sekiyama, Naoyuki Kubota, Tomohide Naniwa, and Joaquin Sitte, Eds. 2006, pp. 314–322, Springer.

[39] Nadir Khessal, "Towards a distributed multi-agent system for a robotic soccer team," in *RoboCup-99: Robot Soccer World Cup III*, Manuela Veloso, Enrico Pagello, and Hiroaki Kitano, Eds., vol. 1856 of *Lecture Notes in Computer Science*, pp. 357–365. Springer Berlin / Heidelberg, 2000.

[40] Hartmut Schmeck and Christian Müller-Schloer, "A characterization of key properties of environment-mediated multiagent systems," in *Engineering Environment-Mediated Multi-Agent Systems: Int. Workshop, EEMMAS 2007. Selected Revised and Invited Papers*, 2008, pp. 17–38.

[41] Li Gao, Shangping Dai, Shijue Zheng, and Guanxiang Yan, "Using genetic algorithm for data mining optimization in an image database," in *Fuzzy Systems and Knowledge Discovery, 2007. FSKD 2007. Fourth International Conference on*, August 2007, vol. 3, pp. 721 –723.

[42] Eugene Lawler, *The Traveling Salesman Problem: A Guided Tour of Combinatorial Optimization*, Wiley, New York, 1985.

[43] Michael Wetter and Jonathan Wright, "Comparison of a generalized pattern search and a genetic algorithm optimization method," in *Proc. 8 th International Building Performance Simulation Association Conference vol III*, 2003, pp. 1401–1408.

[44] R.C Eberhart and J. Kennedy, "Particle swarm optimization," in *Proceedings of IEEE International Conference on Neural Networks*, 1995, vol. 4, pp. 1942–1948.

[45] Fred Glover, "Tabu search - part i," *ORSA Journal on Computing*, vol. 1, no. 3, pp. 190–206, 1989.

[46] Fred Glover, "Tabu search - part ii," *ORSA Journal on Computing*, vol. 2, no. 1, pp. 4–32, 1989.

[47] Felix Dobslaw, "A parameter tuning framework for metaheuristics based on design of experiments and artificial neural networks," in *Proceeding of the International Conference on Computer Mathematics and Natural Computing 2010*. 2010, WASET.

[48] F. T. Lin, C. Y. Kao, and C. C. Hsu, "Applying the genetic approach to simulated annealing in solving some np-hard problems," *IEEE Transactions on Systems, Man, and Cybernetics*, vol. 23, pp. 1752–1767, 1994.

[49] Zbigniew J. Czech and Piotr Czarnas, "Parallel simulated annealing for the vehicle routing problem with time windows," *Parallel, Distributed, and Network-Based Processing, Euromicro Conference on*, vol. 0, pp. 0376, 2002.

[50] Ihor O. Bohachevsky, Mark E. Johnson, and Myron L. Stein, "Generalized simulated annealing for function optimization," *Technometrics*, vol. 28, no. 3, pp. pp. 209–217, 1986.

[51] William L. Goffe, Gary D. Ferrier, and John Rogers, "Global optimization of statistical functions with simulated annealing," *Journal of Econometrics*, vol. 60, pp. 65–99, 1994.

[52] H. Martinez-Alfaro and D.R. Flugrad, "Collision-free path planning for mobile robots and/or agvs using simulated annealing," in *Systems, Man, and Cybernetics, 1994. 'Humans, Information and Technology'., 1994 IEEE International Conference on*, Oct. 1994, vol. 1, pp. 270 –275 vol.1.

[53] Alejandro Quintero and Samuel Pierre, "Assigning cells to switches in cellular mobile networks: a comparative study," *Computer Communications*, vol. 26, no. 9, pp. 950 – 960, 2003.

[54] E. Sundermann and I. Lemahieu, "Pet image reconstruction using simulated annealing," in *Medical Imaging 1995: Image Processing, SPIE Proceedings vol. 2434*, M.H. Loew, Ed., San Diego, California, 1995, pp. 378–386, SPIE.

[55] Maqsood Yaqub, Ronald Boellaard, Marc A Kropholler, and Adriaan A Lammertsma, "Optimization algorithms and weighting factors for analysis of dynamic pet studies," *Physics in Medicine and Biology*, vol. 51, no. 17, pp. 4217, 2006.

[56] Peter A. N. Bosman and Dirk Thierens, "Multi-objective optimization with diversity preserving mixture-based iterated density estimation evolutionary algorithms," *International Journal of Approximate Reasoning*, vol. 31, no. 3, pp. 259 – 289, 2002.

[57] E. K. Burke, S. Gustafson, and G. Kendall, "Diversity in genetic programming: an analysis of measures and correlation with fitness," *Evolutionary Computation, IEEE Transactions on*, vol. 8, no. 1, pp. 47 – 62, February 2004.

[58] Markus Brameier and Wolfgang Banzhaf, "Explicit control of diversity and effective variation distance in linear genetic programming," in *Proceedings of the 5th European Conference on Genetic Programming*, London, UK, 2002, EuroGP '02, pp. 37–49, Springer-Verlag.

[59] Daniel N. Wilke, Schalk Kok, and Albert A. Groenwold, "Comparison of linear and classical velocity update rules in particle swarm optimization: notes on diversity," *International Journal for Numerical Methods in Engineering*, vol. 70, no. 8, pp. 962 – 984, May 2007.

[60] Thomas Back, David B. Fogel, and Zbigniew Michalewicz, Eds., *Handbook of Evolutionary Computation*, IOP Publishing Ltd., Bristol, UK, UK, 1st edition, 1997.

[61] Mitchell A. Potter and Kenneth A. De Jong, "A cooperative coevolutionary approach to function optimization," in *Proceedings of the International Conference on Evolutionary Computation. The Third Conference on Parallel Problem Solving from Nature: Parallel Problem Solving from Nature*, London, UK, 1994, PPSN III, pp. 249–257, Springer-Verlag.

[62] C. M. Fonseca and P. J. Fleming, "Multiobjective optimization and multiple constraint handling with evolutionary algorithms. i. a unified formulation," *Systems, Man and Cybernetics, Part A: Systems and Humans, IEEE Transactions on*, vol. 28, no. 1, pp. 26 –37, Jan. 1998.

[63] C. M. Fonseca and P. J. Fleming, "Multiobjective optimization and multiple constraint handling with evolutionary algorithms. ii. application example," *Systems, Man and Cybernetics, Part A: Systems and Humans, IEEE Transactions on*, vol. 28, no. 1, pp. 38 –47, Jan. 1998.

[64] Ashish Ghosh and Lakhmi C. Jain, Eds., *Evolutionary computation in data mining*, Number 163 in Studies in fuzziness and soft computing. Springer, Berlin [u.a.], 2005.

[65] Claudio Rossi Dept, Claudio Rossi, Elena Marchiori, and Joost N. Kok, "An adaptive evolutionary algorithm for the satisfiability problem," *Evolutionary Computation*, vol. 10, pp. 35–50, 2000.

[66] Pablo Cortés, Luis Onieva, Jesús Muáuzuri, and Jose Guadix, "A revision of evolutionary computation techniques in telecommunications and an application for the network global planning problem," in *Success in Evolutionary Computation*, Ang Yang, Yin Shan, and Lam Bui, Eds., vol. 92 of *Studies in Computational Intelligence*, pp. 239–262. Springer Berlin / Heidelberg, 2008.

[67] James E. Baker, "Reducing bias and inefficiency in the selection algorithm," in *Proceedings of the Second International Conference on Genetic Algorithms on Genetic algorithms and their application*, Hillsdale, NJ, USA, 1987, pp. 14–21, L. Erlbaum Associates Inc.

[68] Tobias Blickle and Lothar Thiele, "A comparison of selection schemes used in genetic algorithms," Tech. Rep., Gloriastrasse 35, CH-8092 Zurich: Swiss Federal Institute of Technology (ETH) Zurich, Computer Engineering and Communications Networks Lab (TIK, 1995.

[69] Tobias Blickle and Lothar Thiele, "A comparison of selection schemes used in evolutionary algorithms," *Evol. Comput.*, vol. 4, pp. 361–394, December 1996.

[70] Jinghui Zhong, Xiaomin Hu, Min Gu, and Jun Zhang, "Comparison of performance between different selection strategies on simple genetic algorithms," in *Computational Intelligence for Modelling, Control and Automation, 2005 and International Conference on Intelligent Agents, Web Technologies and Internet Commerce, International Conference on*, November 2005, vol. 2, pp. 1115 –1121.

[71] David E. Goldberg, *Genetic Algorithms in Search, Optimization and Machine Learning*, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition, 1989.

[72] Melanie Mitchell, *An introduction to genetic algorithms*, MIT Press, Cambridge, MA, USA, 1996.

[73] S. Sumathi, T. Hamsapriya, and P. Surekha, *Evolutionary Intelligence: An Introduction to Theory and Applications with Matlab*, Springer Publishing Company, Incorporated, 1st edition, 2008.

[74] Ling Qing, Wu Gang, Yang Zaiyue, and Wang Qiuping, "Crowding clustering genetic algorithm for multimodal function optimization," *Appl. Soft Comput.*, vol. 8, pp. 88–95, January 2008.

[75] David Levine, "Application of a hybrid genetic algorithm to airline crew scheduling," *Comput. Oper. Res.*, vol. 23, pp. 547–558, June 1996.

[76] K.Q. Zhu, "A diversity-controlling adaptive genetic algorithm for the vehicle routing problem with time windows," in *Tools with Artificial Intelligence, 2003. Proceedings. 15th IEEE International Conference on*, November 2003, pp. 176 – 183.

[77] Ali Kamrani, Wang Rong, and Ricardo Gonzalez, "A genetic algorithm methodology for data mining and intelligent knowledge acquisition," *Comput. Ind. Eng.*, vol. 40, pp. 361–377, September 2001.

[78] Rafal Smigrodzki, Ben Goertzel, Cassio Pennachin, Lucio Coelho, Francisco Prosdocimi, and Jr. W. Davis Parker, "Genetic algorithm for analysis of mutations in parkinson's disease," *Artif. Intell. Med.*, vol. 35, pp. 227–241, November 2005.

[79] Lawrence Jerome Fogel, *On the organization of intellect*, Ph.D. thesis, UCLA University of California, Los Angeles, California, USA, 1964.

[80] D.B. Fogel, "Applying evolutionary programming to selected control problems," *Computers & Mathematics with Applications*, vol. 27, no. 11, pp. 89 – 104, 1994.

[81] David B. Fogel, *System Identification through Simulated Evolution: A Machine Learning Approach to Modeling*, Ginn Press, 1991.

[82] David B. Fogel, "Evolving a checkers player without relying on human experience," *Intelligence*, vol. 11, pp. 20–27, June 2000.

[83] D.B. Fogel, "Applying evolutionary programming to selected control problems traveling salesman problems," *Cybernetics and Systems: An International Journal*, vol. 24, no. 1, pp. 27 – 36, 1993.

[84] Rainer Storn and Kenneth Price, "Differential evolution – a simple and efficient heuristic for global optimization over continuous spaces," *J. of Global Optimization*, vol. 11, no. 4, pp. 341–359, 1997.

[85] K. V. Price, R. M. Storn, and J. A. Lampinen, *Differential evolution: a practical approach to global optimization*, Springer, 2005.

[86] Rainer Storn, "On the usage of differential evolution for function optimization," in *Fuzzy Information Processing Society, 1996. NAFIPS. 1996 Biennial Conference of the North American*, June 1996, pp. 519–523.

[87] Rainer Storn, *Designing digital filters with differential evolution*, pp. 109–126, McGraw-Hill Ltd., UK, Maidenhead, UK, England, 1999.

[88] Feng-Sheng Wang and Horng-Jhy Jang, "Parameter estimation of a bioreaction model by hybrid differential evolution," in *Evolutionary Computation, 2000. Proceedings of the 2000 Congress on*, 2000, vol. 1, pp. 410 –417 vol.1.

[89] René Thomsen, "Flexible ligand docking using differential evolution," in *Evolutionary Computation, 2003. CEC '03. The 2003 Congress on*, December 2003, vol. 4, pp. 2354–2361.

[90] R.K. Ursem and P. Vadstrup, "Parameter identification of induction motors using differential evolution," in *Evolutionary Computation, 2003. CEC '03. The 2003 Congress on*, December 2003, vol. 2, pp. 790–796.

[91] Moez Mnif and Christian Müller-Schloer, "Quantitative emergence," in *IEEE Mountain Workshop on Adaptive and Learning Systems (IEEE SMCals 2006)*, July 2006.

[92] Eric Bonabeau, Marco Dorigo, and Guy Theraulaz, *From Natural to Artificial Swarm Intelligence*, Oxford University Press, 1999.

[93] Steven A. Curtis, Walter F. Truszkowski, Michael L. Rilee, and Pamela E. Clark, "Ants for human exploration and development of space," in *Proc. of IEEE Aerospace Conf.*, 2003, vol. 1, pp. 1–261.

[94] Michael G. Hinchey and Roy Sterritt, "99% (biological) inspiration...," in *Proc. of the 1st IFIP Int. Conf. on Biologically Inspired Cooperative Computing*. 2006, vol. 216 of *IFIP Int. Federation for Information Processing*, Springer.

[95] Roderich Groß, Michael Bonani, Francesco Mondada, and Marco Dorigo, "Autonomous self-assembly in swarm-bots," *IEEE Transactions on Robotics*, vol. 22, no. 6, pp. 1115–1130, 2006.

[96] Roderich Groß, Marco Dorigo, and Masaki Yamakita, "Self-assembly of mobile robots – From swarm-bot to super-mechano colony," in *Proceedings of the 9th International Conference on Intelligent Autonomous Systems*. 2006, pp. 487–496, IOS Press.

[97] Nikolaus Correll and Alcherio Martinoli, "Collective inspection of regular structures using a swarm of miniature robots," in *Proc. of the 9th Int. Symp. on Experimental Robotics*, 2006, vol. 21 of *Springer Tracts in Advanced Robotics*, pp. 375–385.

[98] Nikolaus Correll and Alcherio Martinoli, "Modeling and optimization of a swarm-intelligent inspection system," in *Proceedings of 7th International Symposium on Distributed Autonomous Robotic Systems (DARS 2004)*, 2004, Distributed Autonomous Robotic Systems, Springer Verlag, pp. 369–378.

[99] Srinivas Pasupuleti and Roberto Battiti, "The gregarious particle swarm optimizer (g-pso)," in *GECCO '06: Proceedings of the 8th annual conference on Genetic and evolutionary computation*, New York, NY, USA, 2006, pp. 67–74, ACM.

[100] Xin-She Yang, "Firefly algorithms for multimodal optimization," in *Stochastic Algorithms: Foundations and Applications*, Osamu Watanabe and Thomas Zeugmann, Eds., vol. 5792 of *Lecture Notes in Computer Science*, pp. 169–178. Springer Berlin / Heidelberg, 2009.

[101] D. T. Pham, A. Ghanbarzadeh, E. Koc, S. Otri, S. Rahim, and M. Zaidi, "The bees algorithm, a novel tool for complex optimisation problems," in *In Proceedings of the 2nd International Virtual Conference on Intelligent Production Machines and Systems (IPROMS 2006)*, 2006.

[102] R. Eberhart and J. Kennedy, "A new optimizer using particle swarm theory," in *Micro Machine and Human Science, 1995. MHS '95., Proceedings of the Sixth International Symposium on*, Oct. 1995, pp. 39 –43.

[103] R. C. Eberhart and Y. Shi, "Comparing inertia weights and constriction factors in particle swarm optimization," in *Proceedings of the 2000 Congress on Evolutionary Computation*, 2000, vol. 1, pp. 84 –88 vol.1.

[104] M. Clerc, "The swarm and the queen: towards a deterministic and adaptive particle swarm optimization," in *Evolutionary Computation, 1999. CEC 99. Proceedings of the 1999 Congress on*, 1999, vol. 3, pp. 3 vol. (xxxvii+2348).

[105] Anthony Carlisle and Gerry Dozier, "An off-the-shelf pso," in *Proceedings of the Particle Swarm Optimization Workshop*, April 2001, pp. 1–6.

[106] Zhi-Hui Zhan, Jun Zhang, Yun Li, and Henry Shu-Hung Chung, "Adaptive particle swarm optimization," *Trans. Sys. Man Cyber. Part B*, vol. 39, no. 6, pp. 1362–1381, 2009.

[107] Yuelin Gao and Yuhong Duan, "An adaptive particle swarm optimization algorithm with new random inertia weight," in *Advanced Intelligent Computing Theories and Applications. With Aspects of Contemporary Intelligent Computing Techniques*, De-Shuang Huang, Laurent Heutte, and Marco Loog, Eds., vol. 2 of *Communications in Computer and Information Science*, pp. 342–350. Springer Berlin Heidelberg, 2007.

[108] Yuelin Gao and Zihui Ren, "Adaptive particle swarm optimization algorithm with genetic mutation operation," in *Natural Computation, 2007. ICNC 2007. Third International Conference on*, August 2007, vol. 2, pp. 211 –215.

[109] Michael Meissner, Michael Schmuker, and Gisbert Schneider, "Optimized particle swarm optimization (opso) and its application to artificial neural network training," *BMC Bioinformatics*, vol. 7, no. 1, pp. 125, 2006.

[110] Thomas Kiel Rasmussen and Thiemo Krink, "Improved hidden markov model training for multiple sequence alignment by a particle swarm optimization–evolutionary algorithm hybrid," *Biosystems*, vol. 72, no. 1–2, pp. 5–17, 2003, Computational Intelligence in Bioinformatics.

[111] K. E. Parsopoulos and M. N. Vrahatis, "Recent approaches to global optimization problems through particle swarm optimization," *Natural Computing*, vol. 1, pp. 235–306, 2002, 10.1023/A:1016568309421.

[112] Tao Li, Chengjian Wei, and Wenjang Pei, "Pso with sharing for multimodal function optimization," in *Neural Networks and Signal Processing, 2003. Proceedings of the 2003 International Conference on*, December 2003, vol. 1, pp. 450 – 453 Vol.1.

[113] Changhe Li and Shengxiang Yang, "An adaptive learning particle swarm optimizer for function optimization," in *Evolutionary Computation, 2009. CEC '09. IEEE Congress on*, May 2009, pp. 381 –388.

[114] Walter Cedeño and Dimitris K. Agrafiotis, "Using particle swarms for the development of qsar models based on k-nearest neighbor and kernel

regression," *Journal of Computer-Aided Molecular Design*, vol. 17, no. 2-4, pp. 255–263, 2003.

[115] J. Nenortaite and R. Butleris, "Application of particle swarm optimization algorithm to decision making model incorporating cluster analysis," in *Human System Interactions, 2008 Conf. on*, May 2008, pp. 88 –93.

[116] Marco Dorigo, Vittorio Maniezzo, and Alberto Colorni, "The ant system: Optimization by a colony of cooperating agents," *IEEE Transactions on Systems, Man, and Cybernetics-Part B*, vol. 26, pp. 29–41, 1996.

[117] Ajith Abraham, Crina Grosan, and Vitorino Ramos, *Stigmergic Optimization (Studies in Computational Intelligence)*, Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006.

[118] L. M. Gambardella and M. Dorigo, "Solving symmetric and asymmetric tsps by ant colonies," in *Evolutionary Computation, 1996., Proceedings of IEEE International Conference on*, May 1996, pp. 622 –627.

[119] L. M. Gambardella, ÉD Taillard, and M. Dorigo, "Ant colonies for the quadratic assignment problem," *The Journal of the Operational Research Society*, vol. 50, no. 2, pp. pp. 167–176, 1999.

[120] Bernd Bullnheimer, Richard F. Hartl, and Christine Strauss, "An improved ant system algorithm for the vehicle routing problem," *Annals of Operations Research*, vol. 89, pp. 319–328, 1997.

[121] Ruud Schoonderwoerd, Janet L. Bruten, Owen E. Holland, and Leon J. M. Rothkrantz, "Ant-based load balancing in telecommunications networks," *Adapt. Behav.*, vol. 5, pp. 169–207, September 1996.

[122] Kwang Mong Sim and Weng Hong Sun, "Multiple ant-colony optimization for network routing," in *Cyber Worlds, 2002. Proceedings. First International Symposium on*, 2002, pp. 277–281.

[123] A. Forestiero, C. Mastroianni, and G. Spezzano, "Antares: an ant-inspired p2p information system for a self-structured grid," in *Bio-*

Inspired Models of Network, Information and Computing Systems, 2007. Bionetics 2007. 2nd, December 2007, pp. 151–158.

[124] Kwang Mong Sim and Weng Hong Sun, "Ant colony optimization for routing and load-balancing: survey and new directions," *Systems, Man and Cybernetics, Part A: Systems and Humans, IEEE Transactions on*, vol. 33, no. 5, pp. 560–572, September 2003.

[125] K. Ravikumar and A. Gnanabaskaran, "Aco based spatial data mining for traffic risk analysis," in *Innovative Computing Technologies (ICICT), 2010 International Conference on*, February 2010, pp. 1–6.

[126] Lilia Rejeb, Zahia Guessoum, and Rym M'Hallah, "The exploration-exploitation dilemma for adaptive agents," in *Proceedings of the Fifth European Workshop on Adaptive Agents and Multi-Agent Systems*, 2005.

[127] D. E. Goldberg and R. Lingle, "Alleles, loci and the traveling salesman problem," in *International Conference on Genetic Algorithms and Their Applications*, 1985, pp. 154–159.

[128] I. M. Oliver, D. J. Smith, and J. R. C. Holland, "A study of permutation crossover operators on the traveling salesman problem," in *Proceedings of the Second International Conference on Genetic Algorithms on Genetic algorithms and their application*, Hillsdale, NJ, USA, 1987, pp. 224–230, L. Erlbaum Associates Inc.

[129] Jean-Yves Potvin, "Genetic algorithms for the traveling salesman problem," *Annals of Operations Research*, vol. 63, pp. 337–370, 1996, 10.1007/BF02125403.

[130] G. Reinelt, "The traveling salesman problem: Computational solutions for tsp applications," in *Lecture Notes in Computer Science 840*. 1994, pp. 172–186, Springer-Verlag.

[131] Lester Ingber, "Very fast simulated reannealing," *Mathl. Comput. Modelling*, vol. 12, pp. 967–973, 1989.

[132] Antonio Nebro, Juan Durillo, Carlos Coello Coello, Francisco Luna, and Enrique Alba, "A study of convergence speed in multi-objective metaheuristics," in *Parallel Problem Solving from Nature – PPSN X*, Günter Rudolph, Thomas Jansen, Simon Lucas, Carlo Poloni, and Nicola Beume, Eds., vol. 5199 of *Lecture Notes in Computer Science*, pp. 763–772. Springer Berlin / Heidelberg, 2008.

[133] Sven Tomforde, Ioannis Zgeras, Jörg Hähner, and Christian Müller-Schloer, "Adaptive control of sensor networks," in *Proceedings of the 7th international conference on Autonomic and trusted computing*, Berlin, Heidelberg, 2010, ATC'10, pp. 77–91, Springer-Verlag.

[134] Marco Dorigo and Luca M. Gambardella, "Ant colonies for the traveling salesman problem," *BioSystems*, vol. 43, pp. 73–81, 1997.

[135] Miroslav Benda, Vasudevan Jagannathan, and Rajendra Dodhiawala, "An optimal cooperation of knowledge sources: An empirical investigation," Tech. Rep. BCS–G2010–28, Boeing Advanced Technology Center, USA, July 1986.

[136] Peter Stone and Manuela Veloso, "Multi-agent systems: A survey from a machine learning perspective," *Autonomous Robots*, vol. 8, no. 3, pp. 345–383, June 2000.

[137] M.J. North, T.R. Howe, N.T. Collier, and J.R Vos, "The repast simphony development environment," in *Proceedings of the Agent 2005 Conference on Generative Social Processes, Models and Mechanisms*, 2005.

[138] R. E. Korf, "A simple solution to pursuit games," in *In Working Papers of the 11th Int. Workshop on Distributed Artificial Intelligence*, 1992, pp. 183–194.

[139] Duncan S. Callaway, Mark E. J. Newman, Steven H. Strogatz, and Duncan J. Watts, "Network robustness and fragility: Percolation on random graphs," *Physical Review Letters*, vol. 85, no. 25, pp. 5468–5471, 2000.

[140] Jean-Jacques E. Slotine and Weiping Li, *Applied Nonlinear Control*, Prentice Hall, 1990.

[141] Genichi Taguchi, *Taguchi on Robust Technology Development – Bringing Quality Engineering Upstream*, Amer Society of Mechanical, 1993.

[142] Armin Scholl, *Robuste Planung und Optimierung – Grundlagen, Konzepte und Methoden, Experimentelle Untersuchungen*, Physica-Verlag, Heidelberg, 2001.

[143] Pankaj Jalote, *Fault Tolerance in Distributed Systems*, Prentice Hall, 1994.

[144] Vladimir Shestak, Howard Jay Siegel, Anthony A. Maciejewski, and Shoukat Ali, "The robustness of resource allocations in parallel and distributed computing systems," in *Proceedings of the International Conference on Architecture of Computing Systems (ARCS 2006)*, 2006, pp. 17–30.

[145] Klaus Waldschmidt, "Robustness in soc design," in *Proceedings of the 9th EUROMICRO Conference on Digital System Design: Architectures, Methods and Tools (DSD 2006)*. 2006, pp. 27–36, IEEE Computer Society.

[146] Wilhelm Heupke, Christoph Grimm, and Klaus Waldschmidt, "A new method for modeling and analysis of accuracy and tolerances in mixed-signal systems," in *Proceedings of the Forum on Specification and Design Languages (FDL 2003)*, 2003.

[147] Enrique Alba and Carlos Cotta, "Parallelism and evolutionary algorithms," *IEEE Transactions on Evolutionary Computation*, vol. 6, pp. 443–462, 2002.

[148] J. F. Schutte, J. A. Reinbolt, B. J. Fregly, R. T. Haftka, and A. D. George, "Parallel global optimization with the particle swarm algorithm," *Journal of Numerical Methods in Engineering*, vol. 61, pp. 2296–2315, 2003.

[149] T. Van Luong, N. Melab, and E.G. Talbi, "Parallel hybrid evolutionary algorithms on gpu," in *Evolutionary Computation (CEC), 2010 IEEE Congress on*, july 2010, pp. 1–8.

Appendix

Further Experiments with Noisy Functions

In addition to the experiments presented in Sec. 5.1.3 we have carried out further experiments with 30 different noisy functions provided by the "Black-Box Optimization Benchmarking (BBOB 2010)" environment. Each function is implemented with a moderate and severe level of noise using the Gaussian, Cauchy and uniform distributions[1]. We have investigated the performance of RBI, DE, PSO, GA and SA using 30 and 50 dimensional functions. 30 experiments, each initialised with a different random seed, are carried out and the average best fitness values are recorded. Fig. 7.2 and in Fig. 7.3 show the results provided by RBI, DE, PSO, GA and SA for the functions implemented in 30 and 50 dimensions, respectively. Both results show that RBI can perform effectively in noisy environments provided by BBOB and outperforms PSO, DE, GA and SA. Thus, these results confirm that RBI is more suitable than its competitors to be used for OC systems that work in noisy environments. Please notice that the results shown in Fig. 7.2 and in Fig. 7.3 do not include the optima for the functions. BBOB provides a compiled .dll file (i.e., a black-box), which includes the noisy functions, so that we do not have an access to the concrete implementation of these functions. Since the functions are for example shifted in the given search space, it is not possible to determine the exact optima prior to the optimisation.

[1]For more details on the investigated functions and the corresponding noise models please refer to `http://coco.lri.fr/BBOB-downloads/download10.2/bbobdocnoisyfunctionsdef.pdf`

30 Dim	RBI	DE	PSO	GA	SA
F101	79.48	79.48	79.48	79.48	79.48
F102	79.48	79.48	79.48	79.48	79.48001
F103	79.48443	79.48657	79.51082	79.48473	79.52919
F104	220.85199	152.61075	182.45976	234.3219	172.16333
F105	237.99341	173.45956	189.27661	250.91661	204.96344
F106	206.50398	149.16714	176.224	208.76837	157.58142
F107	96.73843	161.65012	130.61507	134.9848	279.84256
F108	141.64827	198.51077	179.54878	155.07564	299.80164
F109	80.93749	82.09688	85.72365	80.77034	97.23606
F110	512.9628	16200.372	3347.9106	2344.8135	229078.94
F111	6893.7856	81423.766	45214.832	12056.059	308680.3
F112	242.83447	178.50587	222.38193	265.59924	259.7308
F113	166.79283	336.6812	233.98102	402.2199	769.81665
F114	482.49545	594.3805	568.0255	684.9841	1074.0472
F115	108.41792	108.997826	130.9931	119.11111	190.23907
F116	3963.7522	13518.156	8033.228	12690.638	33458.12
F117	18901.082	26693.025	25797.484	27273.729	55366.125
F118	263.4989	-8.582589	1065.0895	488.77972	1651.5576
F119	-46.022324	-36.395462	-41.397552	-37.458794	-14.44014
F120	-34.98048	-25.196583	-27.561552	-27.93768	-0.5877449
F121	-49.677822	-47.59752	-44.654034	-48.657715	-34.34659
F122	-11.861108	-9.234178	-9.913252	-10.946062	-4.852965
F123	-10.101159	-7.63603	-8.335576	-9.152641	-3.7767868
F124	-12.42402	-11.112535	-10.381351	-13.06972	-5.596356
F125	-102.0513	-101.41484	-101.75903	-102.108986	-99.95188
F126	-102.02085	-101.03983	-101.47198	-102.07309	-99.56618
F127	-101.81403	-101.37019	-101.531975	-102.08022	-99.89466
F128	104.25653	112.312675	110.18663	109.55681	120.93715
F129	104.648384	112.57802	111.58498	109.50459	120.59895
F130	51.408276	50.306522	78.61657	50.666206	118.12338

Figure 7.2: The averaged best fitness values obtained by RBI, DE, PSO, EA and SA for the functions with 30 dimensions. Best solutions are shown in grey.

50 Dim	RBI	DE	PSO	GA	SA
F101	79.48	79.48	79.48	79.48001	79.48009
F102	79.48	79.48	79.48	79.48001	79.50602
F103	79.491554	79.49861	79.57847	79.49363	79.643684
F104	219.25842	192.96025	242.43568	236.28128	266.373
F105	219.56903	196.12473	279.96463	292.9958	392.95862
F106	194.18314	165.61641	224.19897	211.59557	194.51306
F107	181.51106	339.62943	278.85706	218.98071	514.1626
F108	233.27792	371.9338	332.08475	252.12299	529.39246
F109	83.92281	87.04368	100.50652	83.28689	124.05528
F110	8615.655	248118.94	60478.234	41310.395	814046.5
F111	61861.375	395346.72	260561.55	77453.82	993149.06
F112	261.25766	203.97769	364.12845	306.63068	391.57303
F113	377.9619	1087.6428	742.0884	715.21204	2027.694
F114	864.16003	1399.4591	1254.7194	1079.9338	2349.1487
F115	138.34337	192.50719	207.37613	157.95592	361.2306
F116	18630.766	66070.875	38711.277	58281.113	140357.75
F117	80338.94	109070.836	99181.6	111589.48	188945.88
F118	359.15976	4231.515	2691.2998	459.26047	3971.3484
F119	-34.829773	-7.4670806	-20.461882	-22.246954	38.649666
F120	-19.053326	8.931076	1.8105683	-11.130099	61.99187
F121	-45.05681	-39.060123	-35.443855	-43.237503	-17.018393
F122	-9.663945	-5.391787	-6.913761	-9.0264225	-1.046232
F123	-8.158922	-3.7420087	-5.182545	-8.157349	0.8243227
F124	-10.603076	-7.875959	-8.176938	-11.348757	-2.9121222
F125	-101.91288	-100.49285	-101.37219	-101.97867	-99.056114
F126	-101.87609	-100.26081	-101.07379	-101.94026	-98.96949
F127	-101.66546	-100.68517	-101.157	-101.94792	-99.044304
F128	112.95254	120.50074	119.0954	114.76153	125.592545
F129	113.240944	121.10119	118.73538	115.095	125.459305
F130	75.877014	110.82847	110.992226	49.165432	124.54994

Figure 7.3: The averaged best fitness values obtained by RBI, DE, PSO, EA and SA for the functions with 50 dimensions. Best solutions are shown in grey.

Lebenslauf

Name:	Emre Cakar
Geburtsdatum:	24.05.1979
Geburtsort:	Gaziantep, Türkei
Staatsangehörigkeit:	türkisch
Schulausbildung:	1990 – 1997 Deutsch-Türkisches Gymnasium Cagaloglu, Istanbul, Türkei Abschluss: Allgemeine Hochschulreife
Studium:	1997 – 2002 Informatikstudium an der Fakultät für Ingenieurwissenschaften der Universität Istanbul, Türkei Abschluss: Diplom Informatiker
Masterstudium:	2003 – 2006 Masterstudium an der Fakultät für Elektrotechnik und Informatik der Universität Hannover Abschluss: M.Sc.-Inf.
Berufstätigkeit:	2006 – 2011 wissenschaftlicher Mitarbeiter am Institut für Systems Enginnering, Fachgebiet System und Rechnerarchitektur an der Leibniz Universität Hannover