Diversity Guided Adaptive Evolutionary Algorithm

Julius Buset Asplin
Nicolay N Thafvelin

Master of Science in Computer Science
Submission date: June 2015
Supervisor: Keith Downing, IDI

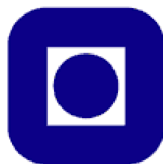
Norwegian University of Science and Technology
Department of Computer and Information Science

Julius Buset Asplin & Nicolay Thafvelin

Diversity Guided Adaptive Evolutionary Algorithm

Master Thesis, Spring 2015

Artificial Intelligence Group
Department of Computer and Information Science
Faculty of Information Technology, Mathematics and Electrical Engineering



Abstract

Parameter tuning in Evolutionary Algorithms (EA) generally result in suboptimal choices of values because of the complex dependencies between the parameters. Furthermore, different scenarios during a run of the EA often have different optimal parameter values.

This thesis aims to better the understanding of how information about previously successful applications of genetic operators can be used to improve the quality of the search by using derandomised self-adaptive parameter control; We utilise the genetic differences between an offspring its parent to adapt a mutation vector. It also explores two different selection strategies that maintains diversity in the population, and the general effect that diversity has on the exploration and exploitation of the solution space.

The adaptive mutation scheme proposed in this thesis has shown to improve the speed of the EA significantly while still being able to solve a wide range of mathematical functions as well as practical problems. Supplemented with a simple scheme that maintains diversity it becomes a more robust implementation well suited for multiple types of problems; especially for problems with computationally expensive fitness tests.

Keywords

Genetic algorithms, derandomised self-adaptation, strategy parameter control, step size control, selection strategy, genetic diversity.

Sammendrag

This is a Norwegian translation of the abstract.
Finjustering av parametere brukt i Evolusjonære algoritmer (EA) resulterer vanligvis i suboptimale valg av verdier grunnet komplekse avhengigheter mellom parametrene. Ulike scenarier under en kjøring av EA har dessuten ofte forskjellige optimale parameterverdier.

Denne avhandlingen tar sikte på å bedre forståelsen av hvordan informasjon om tidligere vellykkede anvendelser av genetiske operatorer kan brukes til å forbedre kvaliteten på søket, ved hjelp selvjusterende parameter kontroll, som baserer seg på forskjellene mellom genotypen til forelder og avkom. Oppgaven utforsker også utvelgingsstrategier som forsøker å opprettholde mangfoldet i populasjonen, og generelt hvordan genetisk mangfold påvirker utforskningen av løsningsrommet såvel som utnytting av gode områder i løsningsrommet.

Den adaptive mutasjonsprosedyren som blir foreslått i denne avhandlingen har vist seg å forbedre hastigheten på EA betydelig, og samtidig være i stand til å løse et bredt spekter av matematiske funksjoner, samt praktiske problemer. Supplert med en enkel ordning som ivaretar mangfold blir dette en robust implementasjon, godt egnet for flere typer problemer; spesielt for problemer hvor testing av individene er ressurskrevende.

Preface

This document was written as the author's master's thesis at the Department of Computer and Information Science at the Norwegian University of Science and Technology, during the spring of 2015. We would like to thank our supervisor, Keith L. Downing, for his guidance throughout our work.

Julius Buset Asplin & Nicolay Thafvelin

Trondheim, June 11, 2015

Contents

1 Introduction **1**
1.1 Background and Motivation . . . 1
1.2 Goals and Research Questions . . . 2
1.3 Research Method . . . 2
1.4 Contributions . . . 3
1.5 Thesis Structure . . . 3

2 Background Theory and Motivation **5**
2.1 Evolutionary Algorithms . . . 5
2.1.1 Genetic Algorithms . . . 6
2.1.2 Genetic Representation . . . 8
2.1.3 Recombination . . . 9
2.1.4 Balancing Exploration and Exploitation . . . 9
2.1.5 Parameter Control versus Parameter Tuning . . . 10
2.1.6 Vector-Based Mutation . . . 12
2.2 Related Work . . . 15
2.2.1 Diversity-Guided Evolution . . . 16
2.2.2 Controlled Vector-Based Mutation . . . 20
2.3 Structured Literature Review Protocol . . . 25
2.3.1 Search Procedure . . . 25
2.3.2 Selection Criteria . . . 25
2.4 Motivation . . . 26

3 Architecture **29**
3.1 Genetic Representation . . . 29
3.2 Reproduction . . . 30
3.2.1 Mutation . . . 30
3.2.2 Recombination . . . 32
3.2.3 Summary . . . 33

3.3 Selection Schemes . . . 33
3.3.1 Parent Selection . . . 34
3.3.2 Distance Measures . . . 34
3.3.3 K-Nearest Neighbours . . . 35
3.3.4 Contribution to Diversity as an Objective . . . 37
3.4 Implementation . . . 39
3.4.1 Genetic Algorithm . . . 39
3.4.2 Graphical User Interface and Data Management . . . 41
3.4.3 Visualisation . . . 43
3.5 Technology . . . 45
3.5.1 Go Programming Language . . . 45
3.5.2 Sails.js Server . . . 45
3.5.3 MongoDB . . . 46

4 Experiments and Results 47
4.1 Test Setup . . . 47
4.1.1 Test Functions . . . 47
4.1.2 Pole-Balancing Problem . . . 52
4.1.3 Robot Arm Function . . . 55
4.2 Experimental Setup . . . 57
4.3 Results . . . 59
4.3.1 Test Functions . . . 59
4.3.2 Convergence Speed and Robustness . . . 60
4.3.3 Different Problem Sizes . . . 62
4.3.4 Rotated Solution Space . . . 64
4.3.5 Pole-Balancing Problem . . . 66
4.3.6 Robot Arm Function . . . 67
4.3.7 Diversity . . . 68
4.3.8 Adaptive Mutation . . . 72
4.4 Contribution to Diversity as an Objective . . . 75

5 Evaluation and Conclusion 77
5.1 Evaluation . . . 77
5.1.1 Robustness . . . 77
5.1.2 Speed of Convergence . . . 81
5.1.3 K-Nearest Neighbours versus Contribution to Diversity . . . 84
5.1.4 Adaptive Mutation . . . 87
5.1.5 Robot Arm Function . . . 88
5.1.6 Pole-Balancing Problem . . . 89
5.2 Conclusion . . . 89
5.3 Future Work . . . 91

5.3.1 Adaptive Rate of Mutation 91
5.3.2 Covariance Matrix 91
5.3.3 Weighing Diversity 91
5.3.4 Phenotype Diversity 92
5.3.5 Steady-State Implementation 92
5.3.6 The Island Model 92

Bibliography 93

A Appendices 1

A.1 Videos 1
A.2 Graphical User Interface 2
A.3 Unique Gene Values 6
A.4 Adaptive Mutation Results 8

List of Figures

2.1 A state diagramming of the EA 6
2.2 An overview of types of parameter settings 11
2.3 The effect on probability density curves of mutation vectors using different parameters 13
2.4 Illustration of diversity-triggered phases, toggling between explore and exploit 16
2.5 Illustration of how solutions can be dominated by others in multi-objective function optimisation 19
2.6 Illustration of DE's utilisation of known good solutions. 26

3.1 Illustrates two scenarios where shortest distance and average distance will lead to different decisions. 38
3.2 State diagram of the GA implementation 41
3.3 Illustration of how the mutation vector, the crossover vector and the parents' genomes are used to construct the offspring. 42
3.4 Pictures from the GUI 43
3.5 Advanced visualisation through the GUI. 44

4.1 Landscape of mathematical benchmark functions 48
4.2 Illustration of how rotation of the solution space can turn a separable problem into a non-separable problem. 52
4.3 Robot arm with four segments 56
4.4 Fitness graphs for four of the benchmark functions 62
4.5 Graphs of the average distance between each individual and its closest neighbour, compared with the respective fitness graphs. . . 69
4.6 Shows the average distance between each individual and its closest neighbour in the population for comparison of the two selection schemes proposed in this thesis. 76

5.1 Illustrates two scenarios where the mutation step size decreases to such an extent that it is impossible for the genes to escape local optimums 78
5.2 Visualisation of Convergence 83
5.3 Illustrates how different fitness landscapes affect the trade-off between fitness and contribution to diversity. 87

A.1 Gui list 2
A.2 Comparing statistics between different GA implementations 3
A.3 Graph tool 4
A.4 Visualisation of pole balancing problem 5

List of Tables

4.1 Definition of the benchmark functions and the range of their parameters . . . 49
4.2 Definition of the benchmark functions and the range of their parameters . . . 50
4.3 Pole balancing variables . . . 53
4.4 Input Parameters for Robot Arm Function . . . 56
4.5 Results from the mathematical benchmark functions . . . 60
4.6 Success rate from the mathematical benchmark functions . . . 61
4.7 Results from running 30 dimensions on some of the mathematical benchmark functions . . . 63
4.8 Success rate from running 30 dimensions on some of the mathematical benchmark functions . . . 63
4.9 Results from running 200 dimensions on some of the mathematical benchmark functions . . . 64
4.10 Success rate from running 200 dimensions on some of the mathematical benchmark functions . . . 64
4.11 Result from the mathematical benchmark functions with rotated solution space . . . 65
4.12 Success rate from the mathematical benchmark functions with rotated solution space . . . 66
4.13 Results from running single and double pole-balancing problem . . . 67
4.14 Success rate for the different pole-balancing tests . . . 67
4.15 Results from running 4-, 8- and 16-segmented robot arm . . . 68
4.16 Success rate and speed from running 4-, 8- and 16-segmented robot arm . . . 68
4.17 Displays statistics on the number of unique gene values throughout runs. . . . 71
4.18 Shows the contributions of mutation and recombination regarding the adaptation of the mutation step size for genes . . . 74

4.19 Success rate and speed of AM-CD in comparison to AM-KN 76

A.1 Results from testing SEA with higher selection pressure on Balancing Pole 5
A.2 Success rate for SEA with higher selection pressure on Balancing Pole 5

Glossary

AI Artificial Intelligence.

ANN Artificial Neural Network.

DE Differential Evolution.

EA Evolutionary Algorithms.

GA Genetic Algorithms.

GP Genetic Programming.

GUI Graphical User Interface.

MSC Mutative Strategy Parameter Control.

PSO Particle Swarm Optimization.

Chapter 1

Introduction

This chapter introduces the work that will be done for this thesis and the motivation behind it. We present the main goal of this thesis and the subordinate questions behind the goal. In Section 1.3 we describe the research methodology applied in our efforts to reach our goal. The succeeding section outlines how this thesis contributes to the scientific community. Finally Section 1.5 gives an overview of how the rest of the thesis is structured.

1.1 Background and Motivation

Evolutionary Algorithms (EA) has received increasing interest over the last decade from both the academic and the industrial fields. EAs perform well in approximating solutions for a wide variety of problems and do so without making assumptions about the underlying problem. This is probably why EA has shown success in many different fields of study, from art and economics to robotics and chemistry.

In many real applications of EAs, a prohibiting factor is the computational complexity. This complexity is due not to EA but to the fitness testing of the problem. Fitness approximation is one method to overcome this, but it is still unclear and theoretical in what way EA can benefit from the approximation model[Jin, 2005]. Another way to reduce computational complexity when fitness testing is expensive is to spend more computational resources on the EA in order to make more educated guesses about what alterations of the genomes to test.

1.2 Goals and Research Questions

In this thesis we focus more on the number of fitness tests needed than on the run time of the EA. We explore whether some alterations to an individual's genome can be considered more likely to produce successful offspring than others, based on information that is available to the EA by default. At the same time we examine ways to maintain diversity and whether an individual's contribution to diversity can help define the successfulness of an individual.

We endeavour to use information about previously successful alterations to a genome to shape the next alterations in a positive way with respect to the probability of producing more successful alterations. The goal can be summarised as followed:

Goal *Explore how information from the evolution of a population's genomes can be used to improve the evolutionary algorithm.*

Research question 1 *How can knowledge about previously successful alterations in each gene be used to improve the quality of future guesses in evolutionary algorithms?*

Research question 2 *How can selection based upon measured contribution to the diversity be used to improve the explorative abilities of evolutionary algorithms?*

1.3 Research Method

The goals and research questions mentioned in the previous section will be addressed by designing and implementing a system which is easily able to run many variations of EA. The EA will run both novel mutation methods, addressed in RQ1, and selection schemes that promote diversity within the population, addressed in RQ2.

We will also implement a Graphical User Interface (GUI) that makes it possible to control multiple systems running the EA implementation. This system will store the results of the EA to permanent storage. Through the GUI it should be possible to compare the results from different runs of the EA, enabling us to compare our EA implementation against similar implementations. We will also create different visualisation methods enabling us to see different aspects of the algorithm when running.

1.4 Contributions

This thesis aims to contribute to a better understanding of how EA's population properties can be utilised to reduce computationally expensive fitness testing.

We can outline the contribution of this thesis in a few key values:

- Better understanding of how adaptive gene-specific mutation can lead EAs to better guesses for each generation
- Better understanding of how information from the population's genomes can be used to maintain valuable diversity in the population
- Better understanding of just how valuable diversity really is and when it is more or less valuable

1.5 Thesis Structure

The rest of this thesis is structured as follows:

Chapter 2 will start with background theory to EA and other disciplines that is necessary to understand for this thesis. We will also present a structured literature review of related work; we will look into similar implementations of the EA. At the end of the chapter, we present the motivation behind the thesis and why the goal and research questions, defined in Section 1.2, are important to address.

Chapter 3 outlines the architecture of the system needed to reach our goals. We also propose an implementation combining and adapting different methods discussed in Chapter 2.

Chapter 4 investigates the current state of the implementation. Here we will also explain the different experiments we have decided to use for testing our implementation and the results we have achieved.

Chapter 5 discusses the results, provides a conclusion and presents some possible future work.

Chapter 2

Background Theory and Motivation

This chapter presents the background needed to understand the content and the related work of this thesis. Section 2.1 gives an introduction to Evolutionary Algorithmss and the key problems encountered when designing such algorithms. The balancing of exploration and exploitation, automated control of parameters and alternative genetic operators are of particular interest to this thesis and are therefore discussed in particular detail throughout the section. Section 2.2 presents similar work done in the field, using techniques such as differential evolution, covariance matrix adaptation and diversity-guided evolution. Finally, in Section 2.3 and 2.4, we discuss the procedure used to find relevant work and the motivation for the thesis.

2.1 Evolutionary Algorithms

Evolutionary Algorithms (EA) refers to a class of algorithms within the field of Artificial Intelligence (AI) that is inspired by biological evolution. EA takes basic concepts and principles from evolutionary biology, like reproduction mutation and survival, and uses them in a search for a good solution for a given problem. EA is adaptable to a wide variety of problems because it makes few assumptions about the underlying problem.

There are numerous subcategories of EA, such as genetic algorithm, evolution-

ary programming, genetic programming and evolutionary strategy. Most of these subcategories were independently developed in the 1990s but share many similarities and were thus grouped under the main topic of EA [Yu and Gen, 2010]. In this thesis we will focus mainly on implementations utilising the genetic algorithm approach.

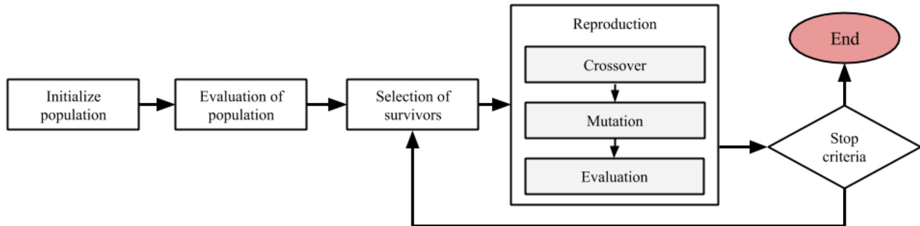


Figure 2.1: A state diagramming of the EA illustrating the main states of an EA and the possible next states.

2.1.1 Genetic Algorithms

A Genetic Algorithms (GA) is an iterative procedure that consists of a candidate solution space which is built up of a population of individuals. Each individual is represented by a set of properties, also called a genome. How the genome is represented is problem dependent, but it is usually represented as a binary vector or a real-valued vector. In Section 2.1.2 a more thorough insight will be given to this topic.

The evolution process starts by initialising the population with randomly or heuristically generated individuals. Then, as seen in Figure 2.1, the population is evaluated by a fitness function, which commonly is the objective function of the optimisation problem being solved. In the next step, individuals with better results from the fitness testing have a higher probability of having offspring. After selection of survivors, the reproduction stage is started. There are many techniques of reproduction, but the most standard is illustrated in Figure 2.1. An offspring is usually created through recombination and mutation of the genomes belonging to the offspring's parents. How many parents the offspring has and how the parents are selected vary for different implementations of GA. A more thorough overview of this topic will be done in Section 2.1.3 The GA terminates after the recombination step if the stop criteria is reached, or a certain predetermined number of generations have been executed. This stop criteria is usually a predefined fitness value that is deemed acceptable. If the stop criteria is not

reached, it continues for another generation, as illustrated in Figure 2.1.

When constructing a GA, there are several design decisions to take into account:

Representation The first thing to consider is how the problem can be represented in the GA and what parameters of the problem should be evolved. For some problems this is trivial. However, for other problems, such as the travelling salesman problem, the representation becomes essential to the success of the algorithm.

Genotype Representation Each individual has a set of parameters which can be mutated and altered; this is called the genotype, or genome. As mentioned earlier, the best representation for the genotype is to a large extent problem dependent, but it is often represented as a binary vector or a real-valued vector.

Phenotype Representation The phenotype is the solution that is applied to the problem. This might be the topology and the synaptic weights of an Artificial Neural Network (ANN) or as simple as input variable values to a complex mathematical function. For many problems there is not a strong apparent distinction between the genotype and phenotype. However, some phenotype conversions are so complex that it is impossible to see intuitively what the phenotype is going to look like merely by looking at the genotype.

Genetic Operators The purpose of the genetic operators is to explore new parts of the solution space that are in some way close to the solutions that scored relatively well during the fitness evaluations. Mutation makes relatively small random alterations to genes in the genome to explore nearby solutions, while crossover combines the genotype of two or more individuals to possibly get the best evolved parts from each of them in the new individual. Both of these operators can be applied in different ways and with different parameters expressing the extent and frequency of their application. For instance, questions like the following might be of interest: How many genes should be mutated? Of what size should the alterations be? How many parents should be combined to create an offspring? And how many points of crossover should be applied?

Fitness Measure The choice of fitness measure can also have a crucial impact on the behaviour of the EA. If, say, you are evolving an ANN for the control of an agent to perform well on several tasks in an environment, the fitness of an individual could simply be the number of tasks it managed to execute. However, since most randomly constructed ANNs would not even begin to approach the task at hand, it is unlikely that the EA would ever find a solution that scored any better than another. Thus, the EA would not

have any preference for which areas of the solution space to examine further. For this reason it is important to have a fitness evaluation that is able to differentiate most solutions in the search space and to do so accurately.

Selection Strategy Last but not least, a proper selection strategy should be chosen. The choice of selection strategy has huge impacts on the behaviour of the EA. Different selection strategies behave differently in different situations with regard to exploration and exploitation. There are a wide variety of different selection algorithms, such as roulette selection and tournament selection. The book "Introduction to Evolutionary Algorithms" by Yu and Gen [2010] further explains how the standard selection algorithms work.

We have directed our focus towards the genetic operators and selection strategies, in other words, the less problem-specific design choices.

2.1.2 Genetic Representation

There are many ways of representing the genetics in GAs, and the main categories are real valued, binary and integer [Rothlauf, 2006]. How the genetics should be represented is to a large extent dependent upon the problem. But as shown through an empirical study of real-valued and binary genetic representation, the EAs that use real-valued genetic representation had more consistent results for each run [Janikow and Michalewicz, 1991]. Real-valued genetic representation also had a better performance, which was argued to be caused mainly by the ability to represent higher precision with less variables in the computer.

In this thesis our main focus will be on real-valued optimisation problems. Using binary code to represent real values has some drawbacks, such as the Hamming Cliff problem [Yu and Gen, 2010]. A Hamming Cliff is when two adjacent values have big hamming distance between them; the neighbouring integers 7 and 8 are a good example. The number of bits that need to be flipped, or mutated, to change a 7 into an 8 when the underlying genetic representation is regular binary code is 4 (from 0111 to 1000), even though in reality they are as close as two integers can be. By contrast, the neighbouring problem refers to the problem that the hamming distance between numbers like 0 (0000) and 8 (1000) is only 1 while the real valued difference between the numbers is 7. Therefore, considering the drawbacks along with other points such as ease of implementation and the ability to easily represent huge continuous spaces of values for each gene, we have chosen to focus on real-valued genetic representation.

2.1.3 Recombination

Many researchers have discussed how recombination of two individuals should be done. In Kauffman [1993] it is also discussed whether recombination of individuals is beneficial at all. Whether recombination is a useful strategy depends on the character of underlying objective function. For problems with a very rough fitness landscape, recombination often gives poor results. Manually tuning the parameters for recombination for each problem is time consuming and difficult. But there are methods for controlling these parameters, which we will come back to in Section 2.1.5.

Several different techniques for recombination of individuals exist. Among these are single-point crossover, uniform crossover and arithmetic crossover. As mentioned in Section 2.1.2, we will focus on real-valued optimisation using real-valued genetic representation. Both Kita et al. [1999] and Deb and Beyer [2001] have defined some useful guidelines for how good real-valued recombination should be performed. Their main concern in these guidelines is to preserve the statistical information of the parents when performing the recombination. It is also important that the offspring should provide as much diversity as possible to the population, but without neglecting the preservation of statistical information. Recombination techniques like single-point and uniform crossover do not fulfil this criterion and are therefore not recommended for real-valued genetic representation [Yu and Gen, 2010].

Arithmetic crossover is one type of recombination scheme that satisfies their guidelines. It works by performing a linear operation that combines two parent genomes to create an offspring. For two parents $X = (x_1, x_2, ..., x_n)$ and $Y = (y_1, y_2, ..., y_n)$, their offspring $Z = (z_1, z_2, ..., z_n)$ is created following Equation 2.1, where α is a random number from the uniform distribution between 0 and 1.

$$Z = \alpha X + (1 - \alpha)Y \tag{2.1}$$

2.1.4 Balancing Exploration and Exploitation

Exploration and exploitation are two important aspects of search problems. Balancing both has proven to be difficult, and this topic has been given much focus, especially in the EA research community [Črepinšek et al., 2013]. A substantial difference between EAs and classic search methods is that EAs maintain and evaluate a whole population of solutions at the same time. The main idea is to

keep individuals from different areas of the solution space that exert promising characteristics. By doing so, there is a higher probability of finding several interesting traits. This will not only broaden the search, but in the process, it might also combine traits from different parts of the solution space, through crossover, to reach even better solutions.

A significant problem for searches in the solution space of approximation problems is local optimums that do not contain satisfactory solutions to the problem. Local and global optimums are points in the solution space in which no small alteration of the genome will provide an increase in fitness. In most searches for solutions to hard approximation problems, the search ends in either a local or the global optimum. Unsatisfactory local optimums pose a threat to the progression of search algorithms, EAs included. For an individual in a local optimum, big and improbable alterations to the genome might be necessary in order to find a competitive solution in terms of fitness.

Maintaining diversity in the population has both proactive and reactive effects. By letting weaker solutions survive, the algorithm can keep searching other parts of the solution space when the best solution so far lies in a local optima. After some generations, the weaker solutions might have led the search to solutions that are better than those in the local optimum, and thus continued progression is achieved. Furthermore, if the entire population resides within a local optimum, then letting weaker solutions survive might increase the probability of breaking out of the local optimum. And it helps if they are further from the point of the local optimum.

A limiting factor to the use of diversity measures in EAs is that it can be very difficult to measure the difference between two individuals. This is especially the case for populations where the size of the genome can vary among individuals [Mattiussi et al., 2004].

2.1.5 Parameter Control versus Parameter Tuning

Tuning of the parameters for EAs is both a time-consuming and difficult task. The issue of controlling the parameters during the run of the EA is therefore an important and promising area of evolutionary computation. Eiben et al. [1999] defined classification rules for different types of parameter control. They defined two major groups for setting the parameters: parameter tuning and parameter control.

Parameter tuning is the manual setting of different parameters before the run of the algorithm. This is usually done by tuning one parameter at a time. It gener-

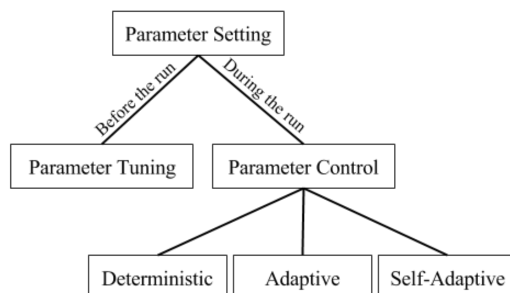


Figure 2.2: Overview of types of parameter control methods for EA (adapted from Eiben et al. [1999])

ally results in suboptimal choices of values because of the complex dependencies between the parameters [Eiben et al., 1999]. Furthermore, different scenarios during a run of the EA often have different optimal parameter values.

An alternative to parameter tuning is to change the parameters during the run of an EA by defining rules from which this change should occur. This is what parameter control aims to do. As illustrated in Figure 2.2, parameter control can be divided into three subgroups, which are listed below.

Deterministic The parameters are adjusted by some deterministic rule. This rule usually contains a time-varying schedule, such as the current generation of the GA, from which it modifies the parameters deterministically. And it does not use any sort of feedback from the GA.

Adaptive The parameters in an adaptive scheme are changed based on feedback from the search, e.g. the quality of solutions, the fitness of each individual relative to the population or diversity within the population.

Self-adaptive Here the parameters have the chance to co-evolve with the individual to which they are applied. The parameters are encoded into the representation of the individual and are included in the recombination and mutation of the individual.

There is a wide variety of EA implementation aiming at controlling all sorts of parameters, such as mutation, recombination and selection strategy. Examples of such implementations that are fairly relevant for this thesis will be investigated in Section 2.2

2.1.6 Vector-Based Mutation

In this section we will investigate different methods that use some type of vector to guide the population through the solution space. The theory of attributing site-specific mutation rates to different sites of the DNA, so-called among-site rate variation [Yang, 1996], has its root in biological models of evolution.

In traditional GAs, one global mutation operator is applied to all genes in the genome. But there are many potential benefits to using more-complex mutation schemes that handle each gene differently. Some of these benefits are listed below.

- The same magnitude of change applied to different genes in the genome might have different effects on the fitness of a solution. For instance, if the goal is to create the best recipe for Christmas gingerbread, a small change in the amount of pepper might have a huge impact on the taste compared to the same change in the amount of flour.
- Genes with different-sized ranges of possible values can make it impossible within certain mutation schemes to choose global mutation parameters that are tailored for all genes.
- Poor knowledge about the solution space might lead a programmer of an GA to set an unreasonable large space of possible parameter values. If the mutation operator uses the size of the range of the parameter to determine the magnitude of change to apply, then unreasonably large ranges might be a problem.

Furthermore, if the mutation parameters in the vector are adapted throughout the run of the algorithm, even more powerful traits can emerge.

- Decreasing the rate or magnitude of change in already fine-tuned genes, and conversely increasing the rate or magnitude of change in genes that exert more flexibility in their values, can provide huge speed-ups for the search. If a gene is particularly fine-tuned, changing it along with the more flexible genes will in all probability have negative effects. Thus, it will dampen or even outweigh the positive effects from other changes in the genome that occurred simultaneously. For instance, when 10 parameters are evolved and 8 of them are fine-tuned within their global optimum, then with a global average step size for change through mutation, all changes done to the already fine-tuned genes will lead to decreases in fitness values. This happens even if the two nonoptimal parameters are changed to their optimal. This scenario makes it difficult for the last parameters to evolve to their optimal values.
- Usually when GAs are employed, the optimisation problem is nonseparable,

meaning that the effects of a change in one parameter cannot be known without the context of the other parameters. Otherwise, if it is known that the problem is separable, then the search could be divided, separated into several searches, one for each gene, and the solutions simply combined afterwards. Figuring out which parameters correlate more with each other and mutating these simultaneously might be beneficial.

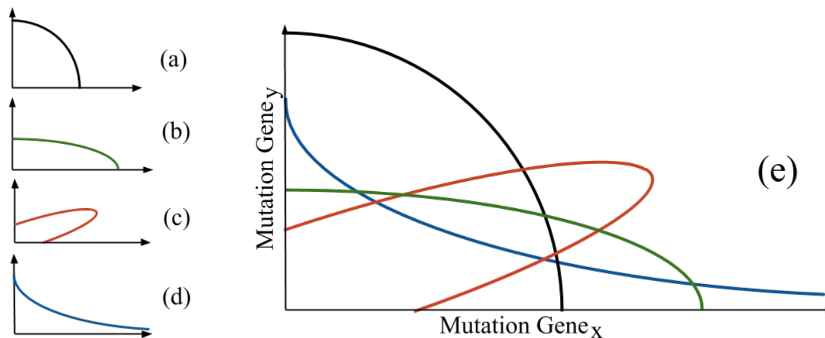


Figure 2.3: Roughly sketches the effects of using different types of mutation. The lines present outcomes of equal probability in different scenarios using different types of mutation. The x- and y-axis denote the magnitude of change done to each gene. (a): One global mutation parameter. (b): Independent mutation parameters for each gene. (c): Covariance parameters and positive covariance between the mutation sizes. (d): Covariance parameters and negative covariance between the mutation sizes. (e): All scenarios in the same plot to show the potential benefits of using more parameters to specify the sort of change that is likely to be successful.

An obvious upgrade from a single adaptive parameter for the individual as a whole is to have one or more free mutation parameters for each gene [Ostermeier et al., 1994; Vafaee and Nelson, 2010]. The parameters for each gene can be seen as an n-dimensional mutation vector for the genome, where n is the number of genes being evolved. A further extension would be to include a complete covariance matrix that also tracks the way each gene evolves relative to each other gene [Hansen and Ostermeier, 2001]. Genes that are in no way correlated will only distort each other's feedback from fitness testing if they are all changed simultaneously. In the extreme case where no genes are concerned with the state of the others, the natural thing to do is to separate the problem into several approximation problems, one for each parameter that is to be evolved. However, since GAs are applied to the search problem, one can assume that the optimal value of every parameter is dependent on the state of at least one other parameter. From this follows an interest in figuring out which parameters are more correlated

than others.

Figure 2.3 illustrates how the different mutation schemes mentioned above can affect the mutation of two genes. The figure shows the shape of lines of equal probability of occurrence in different scenarios. As illustrated by Figure 2.3(a), with a single global mutation step size, the genes have equal probability of mutating by any given amount, and the amount by which a gene is changed is independent of the amount by which other genes are changed. Figure 2.3(b) is an example of a curve of equal probability when each gene has its own mutation parameter for the expected size of a mutation. Mutations to $Gene_x$ are likely to be much bigger in size than mutations to $Gene_y$. Finally, Figure 2.3(c) and (d) are examples in which parameters regarding the covariance between the two step sizes are defined. For Figure 2.3(d) for instance, if $Gene_x$ is changed by a lot, then the probability of $Gene_y$ being changed by a lot is significantly decreased. In Figure 2.3(c), the opposite is the case. If either (c) or (d) are correct in their approximation of what combination of mutation sizes is likely to be most successful, then the benefit from using such detailed specifications is huge.

Differential Evolution

Differential Evolution (DE) is a subcategory of GAs which uses the distance between two arbitrary individuals in the population to construct mutation vectors. DE was suggested in 1997 by Storn and Price [1997], with the aim of being easy to use, requiring few control parameters and also being robust. DE is used mainly for multidimensional real-valued optimisation functions because of the required real-valued vector structure of the population.

The main difference of DE compared to other GAs lies in the recombination and mutation operators. DE utilises site-specific mutation through the use of a mutation vector. The mutation vector is generated by exploiting information retrieved from the genome of multiple individuals in the population.

During the reproduction stage, a mutation vector, mut_i, is generated for each individual, $x_1, x_2, ..., x_i$. The mutation vector for individual x_i is generated by creating a simplex from three random individuals in the population, $r_1, r_2, r_3 \in \{x_1, x_2, ..., x_i\}$, where $r_1 \neq r_2 \neq r_3 \neq x_i$. From these three individuals, the mutation vector is then defined by Equation 2.2, where F is a real and constant factor which usually resides in the range between 0 and 1.

$$mut_i = r_1 + F * (r_2 - r_3) \tag{2.2}$$

After the mutation vector is generated, a uniform crossover is performed, with a constant crossover probability $CR \in [0, 1]$. But unlike most other GAs, the recombination is not done with other individuals; instead it is performed with the mutation vector generated for that individual. In other words, an offspring, o_i, is created by uniform crossover operation between mut_i and x_i. The new offspring is inserted into the population only if it outperforms its parent, x_i, in fitness score. If it does not outperform its parent, then the offspring and mut_i are disposed of, and the parent x_i remains untouched.

Particle Swarm Optimisation

Particle Swarm Optimization (PSO) belongs to swarm intelligence of AI, which is the collective behaviour of decentralised and self-organised systems. The algorithm was originally developed to simulate social behaviour by representing the movement of organisms. But the algorithm was simplified by Eberhart and Kennedy [1995] so it could be used for optimisation tasks.

The basic PSO works by having a swarm of moving particles that explores the search space. The movement of a particle is defined by a simple formula, inspired by the movement of real particles, which gives each particle a velocity. In the updated version of PSO, inertia was also included for particles [Shi and Eberhart, 1998]. In contrast to real particles, the movement formula also takes into account the best-known position that each particle as visited and the best-known position in the whole swarm. The result is that particles are attracted by each other, their own best experienced solution, and the best position that is known overall.

PSO shares many similarities with the GA. Both utilise population-based search methods. Both methods also move from a set of points in the search space to another set of points in an iterative procedure. But in contrast to GAs, the particles in PSO do not reproduce. You could say the particles only "mutate" in the form of moving through the search space according to their velocity vector.

2.2 Related Work

In this section we will inspect related work done in the same research area as addressed in the research questions defined in Section 1.2. In the first section, we will investigate research done using some type of diversity measure for balancing the exploration and exploitation of population. In Section 2.2.2 we will explore different parameter-controlled EA implementations using different types of vector-based mutation.

2.2.1 Diversity-Guided Evolution

Diversity in the population is definitely one of the key issues for the performance of EAs, mainly to avoid premature convergence. Most selection algorithms introduce a random factor to maintain diversity in the population, such as tournament selection and roulette wheel selection. But few algorithms endeavour to guide the population using diversity. The measure of diversity is usually used only as a parameter to analyse how EAs perform [Ursem, 2002]. In this section we will investigate three papers which use some type of diversity measure in different ways to maintain diversity in the population.

Diversity-Triggered Phases

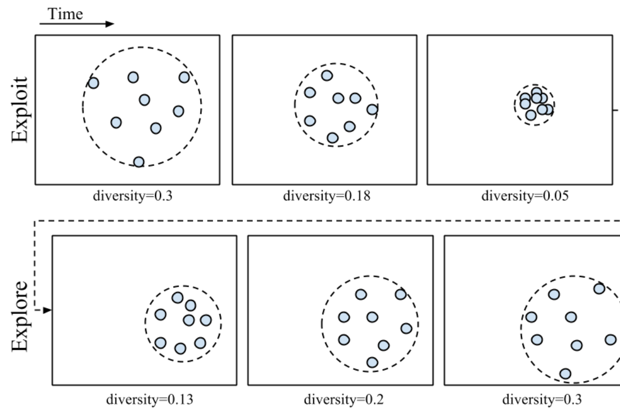


Figure 2.4: Illustration of diversity-triggered phases, toggling between explore and exploit depending on the diversity in the population. This figure is adapted from Ursem [2002]

Ursem [2002] has created an extension to the EA which can be used for multiple types of EAs and is fairly simple to implement. The extension aims to prevent the EA from premature convergence by alternating between two phases, explore and exploit. The alternation between the phases is triggered by a diversity measure of the population. The diversity measure used by Ursem [2002] is the distance from each individual to an average point of all individuals in the solution space. The average point consists of the average values of each gene individually. This is not the most accurate measure of diversity in the population, but it requires very little computation compared to other diversity-measuring techniques [Ursem, 2002].

The two phases of the algorithm are illustrated in Figure 2.4. The algorithm starts by running the EA in the usual manner; this is called the exploit phase. As time passes, the population converges to a local optimum, causing the diversity to decrease. When the diversity reaches a given threshold, $diversity_{low}$, the algorithm switches to the explore phase. The explore phase mutates the individuals' genomes until the diversity in the population reaches a given diversity threshold, $diversity_{high}$. This phase does not require any fitness testing; it only calculates the distance-to-average point after each mutation cycle. The implementation also uses a vector-based mutation to speed up the explore phase, which we will look at in Section 2.2.2.

Their extensions were added to a standard GA implementation, and it outperformed other well-known non-diversity-guided GA implementations on multimodal optimisation functions. The explore phase prevented premature convergence to some extent, but it was difficult to choose appropriate diversity thresholds. They mentioned in further work that deterministic parameter control on $diversity_{low}$ and $diversity_{high}$ would make this simpler.

Diversity Measure for Replacement Scheme

Lozano et al. [2008] use a rather different approach to maintain the diversity within the population. They propose a replacement scheme in a steady state EA. Steady state EAs differs from standard EAs in that there are no generation steps. In a steady state EA, each new individual competes on its own against the current population. The replacement scheme determines whether an individual should be included in the population and which individual it replaces.

Lozano et al. [2008] proposed a replacement scheme in which individuals whose contribution to diversity is relatively small are more likely to be replaced than those with a higher contribution to diversity. Whenever a new individual is under evaluation, the set, D, of all individuals within the population whose fitness is worse than the new individual's, is created. If the individual in D with the lowest contribution to diversity has a lower contribution to diversity than the new individual, then the new individual takes its place. Otherwise, the new individual simply replaces the individual in D with the worst fitness, given that the set is not empty. The contribution to diversity is given by the distance (e.g. Hamming distance, Euclidean distance) between the individual and its nearest neighbour.

In this way the replacement scheme strives to remove solutions that have poor scores in both fitness and contribution to diversity. However, the reward for having a greater contribution to diversity can be seen as a delayed execution.

Contribution to diversity merely determines the order in which individuals are replaced, giving diverse solutions more time to explore their uniqueness before they are replaced.

As we mentioned in Section 2.1.4, a weak solution with high diversity may lead to finding a better optimum after some generations. Lozano et al. [2008] explain that the reason for kicking out the worst solutions even when their contribution to diversity is higher is the increased ability to converge: "For most practical problems, the refinement of the best regions of the search space (i.e. the convergence) requires the loss of population diversity." [Lozano et al., 2008]

Their implementation performed significantly better than the other diversity-guided implementations they used for comparison. But when we compared the results with that of Ursem [2002], mentioned above, we see significantly better results on the part of Ursem [2002]. We cannot be sure whether this outperforming is caused by a better method of maintaining diversity, as it may well be caused by the structural difference between steady-state and generation-based EA, differences in the genetic operators or different parameter settings.

A somewhat similar but much more minimalistic approach to maintaining diversity is deterministic and probabilistic crowding. Probabilistic crowding was introduced by Mengshoel and Goldberg [1999]. The key point is to maintain diversity by having individuals compete with the individuals that are the most similar to themselves.

Diversity as Objective

To the best of our knowledge, de Jong et al. [2001] were the first to use diversity as an objective in a multiobjective EA. In multiobjective EA, the proposed solution is tested on more than one evaluation function, i.e. objective functions. The results from these functions are combined in some way to determine the overall value, i.e. fitness of the individual. de Jong et al. [2001] proposed using the diversity gain from adding an individual to the population as an objective in their multiobjective approach. They used the average squared distance to the other population members to represent the diversity gain.

In addition to using diversity gain as an objective, they follow up by removing all dominated individuals from the selection. An individual is dominated if another individual has a better or equal score on all objective functions. The truth statement below, 2.3, expresses the state of the population after the selection phase. For every two individuals within the population, there exists an objective function for which the first individual has a better score and one for which the other has a better score.

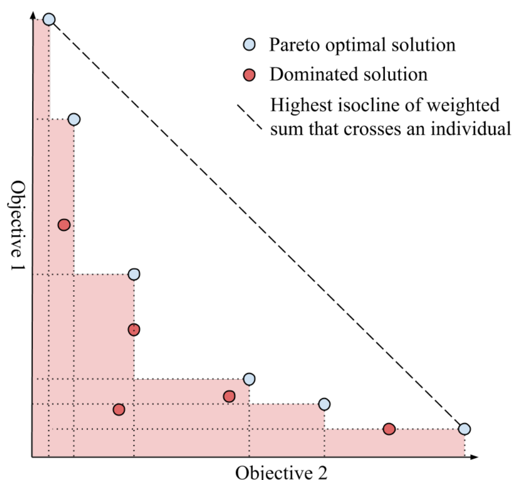


Figure 2.5: Illustration of how solutions can be dominated by others in multiobjective function optimisation. This figure is adapted from de Jong et al. [2001]. Each axis represents the score on an objective function, and each point represents an individual's scores.

$$\forall x, y \in P : \exists f \in OF : f(x) > f(y) \wedge \exists f \in OF : f(y) > f(x) \tag{2.3}$$

Figure 2.5 shows examples of dominated individuals. In this particular illustration, there is a negative correlation between the score on one objective and the score on another. This is not always the case, but it is particularly likely when diversity and fitness are the two objectives. It follows naturally from the fact that increased fitness in a region will lead to more surviving offspring in that same region, which in turn decreases the contribution to diversity in that particular region.

de Jong et al. [2001]'s multiobjective approach was used in Genetic Programming (GP), which is a specialisation of GA. In GP each solution is usually represented by a tree structure that can vary in size. Consequently, calculating the distance between individuals is done quite differently than in real-valued GA, where the size of the genome remains fixed throughout each run of the algorithm.

However, the desired behaviour is the same. de Jong et al. [2001] reported that when large portions of the population are centred in a small area of the fitness landscape, then a potential for scoring well on the diversity function dissipates the population. Conversely, when individuals with the same score in diversity have

similar scores in fitness, the individuals that merely scored well on the diversity function are removed from the population. When some area is discovered to have quite good fitness values, more offspring survive in this region, and thus the gain from diversity in the region decreases.

In their results, de Jong et al. [2001] saw that much smaller population sizes could be applied because of the diversity objective, and the algorithm needed significantly fewer evaluations than regular GP without their added objectives, which, in addition to diversity, included an objective for minimising the size of the tree.

2.2.2 Controlled Vector-Based Mutation

In Section 2.1.6 we investigated the theory behind vector-based mutation. In this section we will inquire into research done in this field which also used some type of parameter control.

Adaptive Probabilistic Mutation Vector

Inspired by the among-site mutation rates from the biological model, Vafaee and Nelson [2010] created an adaptive scheme for their mutation operators. Instead of randomly flipping bits, their mutation operator specified different mutation rates for different genes in the individuals. These gene-specific rates were derived from the highly fit individuals in the population in a probabilistic manner, creating a global mutation vector which worked as an attractor for the mutation of different genes. This implementation had decent results. It performed better than the other implementations they compared it to, both with better scores and lower standard deviations.

The Markov chain of this implementation was later derived, which proved the convergence of their implementation [Vafaee et al., 2014]. A Markov chain is a state space that undergoes transitions between the states using a probabilistic trajectory, where the next state depends only on the current. It is natural to view GA as a Markov chain, considering the current population as the current state. The Markov chain of Vafaee and Nelson [2010] was derived using a well-known Markov chain analysis model for GA [Nix and Vose, 1992]. This method works well because it makes no assumptions restricting the population size or the trajectory of the Markov chain.

Adaptive Average-Point Mutation

In Section 2.2.1 we investigated Ursem [2002]'s diversity-guided EA. In that work a very simple adaptive mutation vector is used in the exploration phase. Instead of random mutation in the exploration phase, the average point in the population is exploited. The mutation vector of an individual is created by creating a vector phasing in the opposite direction of the average point. Logically this speeds up the exploration phase drastically, but it is difficult to say whether this improves the end results. Ursem [2002] did not report how their implementation performed without this feature.

It is also worth mentioning the adaptive mutation scheme used by Srinivas and Patnaik [1994], which is especially good at exploring local optimums while maintaining some level of exploration. In their approach, the basins of attraction are subject to much disruption, while the attractor points are subject to less disruption. So the individuals at the basins of attraction are mutated to find possible new paths that lead to other optimums while individuals near the attractor point are mutated to find the exact point of attraction. Finding out which individuals are at the basins of attraction, and which are not, is not trivial. Srinivas and Patnaik [1994] simply uses the fitness of each individual relative to the best-achieved fitness so far to make this distinction. Though it is an old paper, the results were good compared to the standard genetic algorithm, and we find the idea behind it very interesting.

Self-Adaptive Differential Evolution

DE uses a simplex-based vector for the mutation of each individual, as we explored in Section 2.1.6, and the low number of input parameters used in DE makes it a good candidate for parameter control. As mentioned in Section 2.1.6, DE requires only two parameters: the crossover rate, CR, and the fraction, F, of the difference between two individuals to use when when generating the mutation vector. Brest et al. [2006] proposed a method making both of these parameters, CR and F, self-adaptive. In their implementation, each individual has its own values for F and CR which are used during the creation of their offspring. The parameter values are then passed on from parent to child, but there is a given probability of mutating the CR and F. The parameter values for the child, F_i^{g+1}, is defined as follows:

$$F_i^{g+1} = \left\{ \begin{array}{ll} F_l + r_1 F_u & \text{if } r_2 < \tau_F \\ F_i^g & \text{otherwise} \end{array} \right\}$$

$$CR_i^{g+1} = \left\{ \begin{array}{ll} r_3 & \text{if } r_4 < \tau_{CR} \\ CR_i^g & \text{otherwise} \end{array} \right\} \quad (2.4)$$

$r_1 ... r_4$ are taken randomly from the uniform distribution $[0, 1]$, τ_F and τ_{CR} are user defined probabilities of mutating F and CR, and F_l is the lower and $F_l + F_u$ is upper bound of F; $F \in [F_l, F_l + F_u]$. By using this technique the probability of using the same parameters during the creation of the child as was used during the creation of the parent is $(1 - \tau_F)$. Consequently, once good a value is deemed successful through the survival of an offspring it will continue applying the same value under the creation of its offspring, though with some exploration of arbitrary different values.

Standard DE already performs very well on mathematical optimisation problems, and with this simple extension of DE, the results were improved. Furthermore, this implementation has the advantage of having no parameters which need to be tuned for different problems.

Another approach using the adaptive scheme for DE is the fuzzy adaptive DE by Liu and Lampinen [2005]. The mutation parameter, F, and the crossover parameter, CR, are adapted using a fuzzy logic control approach. The implementation exploits the fitness values and genetic information from the current population to adapt F and CR. The fitness values and genetic information are depressed and fed to a fuzzy logic controller (FLC), which maps input to output using different membership functions.

This is a very complex approach to achieve adaptation, especially in comparison to the self-adaptive DE. It performed well compared to the standard DE, especially for higher-dimension problems. However, the self-adaptive DE looks even more promising in its results.

Strategy Candidate Pool

Qin et al. [2009] and Mallipeddi et al. [2011] extended self-adaptation in DE to include strategy candidate pools. In DE, several different techniques can be applied to generate mutation vectors.

The standard approach is to pick three random individuals from the population and use them to generate the mutation vector. However, this makes for a huge space of possible vectors; Some of these vectors might be more likely to produce successful offspring than others. Other approaches include always using the best

individual combined with one randomly picked individual, or using only a single random individual or the best individual as the mutation vector.

These different approaches have different characteristics and have their raison d'etre in different states of a search. Qin et al. [2009] and Mallipeddi et al. [2011] kept different strategies in a candidate pool and measured their respective success rates in succeeding generations. The ones that had a higher success rate in previous generations were favoured for the generation of new trial vectors in succeeding generations.

The most defining difference between the two is that the Qin et al. [2009] simply updates the possibility of being assigned each strategy for all individuals, while in the latter, each strategy and control parameter has an equal probability of being chosen. If a specific strategy is able to produce an offspring that scored better than its parent, then this offspring will try the same strategy again to produce its own children in the succeeding generation. While the approach used by Qin et al. [2009] did somewhat better than that of Brest et al. [2006] for problems of 10 and 30 dimensions, the implementation in Mallipeddi et al. [2011] outperformed both on 50 dimensions of the test functions.

GA-PSO

Some efforts have been made to combine GA and PSO; among these are Kao and Zahara [2008]. In their hybrid algorithm, an GA works in parallel with a PSO in each generation. At the start of a generation, the population is split in two by their fitness. The GA is assigned the more fit individuals while the PSO is assigned the less fit half of the population. The idea is to combine a PSO's swift convergence with a GA's ability to explore and escape local optimums.

We include this as a related work because PSO has a characteristic, its speed, which is similar to that of vector-based mutation. Kao and Zahara [2008] achieved good results in terms of success rate on many functions.

However, the biggest problem size that was tested on was 10 dimensions of the Rosenbrock test function with a small range for each gene. In terms of efficiency with respect to the number of fitness evaluations needed, the hybrid algorithm seemed to have the same characteristics as regular PSO.

Derandomised Self-Adaptive Mutation

Ostermeier et al. [1994] introduced a completely derandomised method of doing Mutative Strategy Parameter Control (MSC). This derandomised approach was

further extended to adapt a complete covariance matrix in Hansen and Ostermeier [2001].

In their paper they point out that selection of strategy parameter settings in MSC is indirect, and thus, because of the highly stochastic outcome of applying a specific mutation parameter setting for one generation, the selection process for parameter settings will be highly disturbed.

In other words, since the same step size can be produced from virtually any standard deviation, the success of a change in the standard deviation parameter cannot be based on the success of a single sample from it. With greater change rates for the parameter settings, the underlying information is increasingly stochastic and thus unreliable.

At the same time, it is beneficial to have a high change rate on both the strategy parameter level and the actual problem parameters in order to achieve a swift adaptation. This creates a trade-off between swift adaptation and the validity of the selection process that is difficult to optimise. Consequently, the possible change rate of each strategy parameter decreases with the number of strategy parameters that are being adapted.

In order to ensure the use of valid information when adapting the mutation parameters, Ostermeier et al. [1994] use the realised mutation step sizes that led to fit individuals to approximate the ideal standard deviation for the mutation. This technique can be applied to determine the global step size parameter and the step size for each gene in the solution and to construct a complete covariance matrix that holds values for the covariance between all pairs of genes. The latter demands a great number of free strategy parameters, $\frac{n^2-n}{2}$.

The core concept of their algorithm is simple. Comparing the realised mutation step size for selected individuals to the expected mutation step size given the previous settings can offer some insight as to whether the standard deviation should be increased or decreased. For a single global step size, the length of the entire vector of individual step sizes for each gene is compared to its expected length. In the case of each gene-specific mutation step size, σ_i, independently, the expected deviation is $\sigma_i\sqrt{\frac{2}{\pi}}$[Ostermeier et al., 1994].

Hansen and Ostermeier [2001] found that this derandomised approach for covariance mutation matrix adaptation was able to adapt more efficiently to nonseparable problems than its randomised counterpart and the derandomised approaches without covariance parameters. For problems with huge differences in sensitivity for each axis (gene), gene-specific mutation parameters proved to have huge benefits. Hansen and Ostermeier [2001] also discuss many aspects of adaptation speed and the liability of the information received in different adaptation processes.

2.3 Structured Literature Review Protocol

This section describes the structured literature review protocols used to find literature for this thesis.

2.3.1 Search Procedure

We defined a search procedure to enhance the efficiency and the quality of the results when searching for relevant research. We decided to utilise a given set of tools and defined a list of keywords which proved to give good results.

Search Keywords "diversity", "parameter control", "adaptive", "vector mutation", "differential". Most of the search keywords were used in combination with evolutionary computation types, such as "evolution", "evolutionary algorithms", "genetic algorithm", "genetic programming", "evolutionary strategies".

Following is a list of the search tools which were used to find appropriate research.

Google Scholar Used for search.

Wikipedia Sources of articles on Wikipedia were used to find relevant papers.

Survey paper references For instance, we used references from Črepinšek et al. [2013].

Introduction to Evolutionary Algorithms (2010) One of the most popular modern books on evolutionary algorithms, by Yu and Gen [2010].

2.3.2 Selection Criteria

There is a vast amount of research done in the field of EA, and selecting the right papers was a difficult procedure. We therefore defined some criteria that we required the papers to fulfil, which can be summarised as follows:

- Released in the last 20 years. Exceptions made for articles concerning the origin of a topic.
- Published by a trusted publisher, such as Springer, IEEE and MIT Press Journals.
- Cited by at least 10 other publications. Exceptions made for recently published papers given they were published by a trusted publisher.

2.4 Motivation

In this thesis we wish to combine and adapt existing approaches that have been discussed throughout this chapter to produce a traditional genetic algorithm capable of solving a wide range of problems with great speed. Through our search for relevant literature, we found that our questions have been asked before, and several different proposals have been made. However, we see room for a novel technique which combines adaptive vector mutation with promotion of diversity.

One important aspect of GA which we take interest in is the mutation of individuals. It has been shown that a fixed mutation rate can be expected to be suboptimal when balancing exploration and exploitation [Cervantes and Stephens, 2009]. A special interest lies in the utilisation of a mutation vector, inspired by DE, to guide the individuals in a direction of the search space. Figure 2.6 shows how DE utilises existing information in the population to create a mutation vector, instead of doing random guesses, which is the default for standard GA. The information used by DE is known to produce good individuals, at least for a specific point in the solution space, and this information is used to create new individuals; the vector that is added to a random individual to make up the vector that is used during crossover is the exact difference between two known good solutions. Notice in Equation 2.2, that if $r_1 = r_3$ and $F = 1$, then $mut_i = r_2$. That is to say that the result would be a solution which is known to be quite good. If $0 < F < 1$, then mut_i would be somewhere on the direct line between r_1 and r_2. By applying a vector that is known to be good for one of the existing individuals to any other of the existing individuals, the rate of success is increased. This technique seems to work, but it differs slightly from what we were looking for in the sense that information about successful mutations are used indirectly.

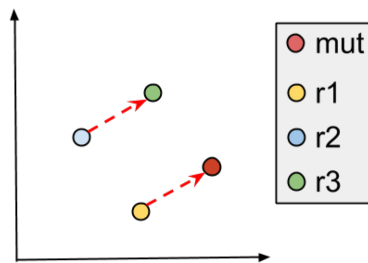


Figure 2.6: Illustration of how DE utilises information from the populations genomes to produce new individuals. r_1, r_2 and r_3 are the individual solutions used to produce mut_i (mut) following Equation 2.2. The x- and y-Axis are simply values for two arbitrary genes in their genomes.

PSO utilises information about the best solutions found thus far to perform a vector-based mutation in a direct and deterministic way through the speed and acceleration of each particle. Although this is not at all based upon previously good mutations, but rather on the direction of good solutions, we believe in the idea of letting each particle do its own exploration with its own inertia.

We take special interest in the adaptation process proposed by Hansen and Ostermeier [2001], as it gives a very direct answer to our first research question. However, while they view the entire population as a single search, we endeavour to uphold the search behaviour of the more traditional GA, where individuals can be viewed as conducting their own search and competing with others to find the best solutions.

As we have mentioned earlier, balancing exploration and exploitation of GA is of particular interest in this thesis. Throughout this chapter we have investigated different methods that exploit information from both the search space and the population to improve future guesses and maintain the diversity in the population, and they all showed promising results. Furthermore, truly adaptive mutation rates will inevitably result in convergence to an optimum when fitness is the only basis for selection. In order to avoid premature convergence, some maintenance of diversity is imperative.

Chapter 3

Architecture

In this chapter, we put forth a GA with a novel mutation and selection scheme which combines and adapts different methods discussed in Chapter 2. We also explain the implementation of the system and the technologies we have used. We start out by presenting the genetic representation in Section 3.1. In Section 3.2 an explanation of the reproduction scheme of the GA is exhibited. The genetic operators we have used and the motivation for using these operators will be presented here. Following this, the two selection schemes which promote diversity in the most direct way will be presented in Section 3.3. Section 3.4 will present the actual implementation of the GA, along with the system used to test different implementations and various analytical visualisation tools. And finally, Section 3.5 presents an overview and a short explanation of the technologies used for this thesis.

3.1 Genetic Representation

Because of the drawbacks with binary representation mentioned in Section 2.1.2, and because of the nature of our approach, we have chosen real-valued genetic representation. Furthermore, we have limited our research to fixed-sized genomes because of the difficulty of measuring diversity among individuals with different-sized genomes mentioned in Section 2.1.4. In general, our focus is on solving continuous functions for which possible solutions can be represented genetically by N-dimensional vectors.

3.2 Reproduction

The different operators and their respective adaptation schemes used under reproduction are presented in this section. In summary, we apply a Gaussian mutation combined with probability of mutation and N-point crossover. However, one free strategy parameter is assigned to each gene, and this parameter partially determines the size of the mutations done to this gene. These parameters are adapted using a derandomised adaptation scheme, similar to the scheme used by Hansen and Ostermeier [2001], which was examined in Section 2.2.2.

3.2.1 Mutation

Our mutation scheme is presented in Equation 3.1. We use a probability of mutation, so only a fraction of the genes will undergo the Gaussian mutation during a single reproduction.

$$\begin{aligned} X_{o,g} = & \left\{ \begin{array}{ll} X_{p,g} & \text{if } r \leq P_m \\ X_{p,g} + M_g & \text{otherwise} \end{array} \right\} \\ M_g = & \sigma_{p,g} n_{g,1} n_{g,2} \end{aligned} \tag{3.1}$$

$X_{o/p,g}$: The value of gene g, for offspring (o) / parent (p).

$r \in \mathcal{U}(0,1)$: Value taken randomly from the uniform distribution between 0 and 1.

P_m : User-defined global value stating the probability of mutation for all genes.

$n_{g,i} \in \mathcal{N}(0, \sqrt{\frac{\pi}{2}})$: Values taken randomly from the Gaussian distribution with mean 0 and standard deviation $\sqrt{\frac{\pi}{2}}$.

$\sigma_{p,g}$: Adaptive strategy parameter stating the average size of mutations done to the parent's (p) gene g.

The probability of mutation, P_m, determines the likelihood of mutating a gene during reproduction. Having $P_m = 0.1$ would result in approximately 10% of the genes in the genome being mutated when creating a new individual. And the standard deviation for the number of genes that are mutated would be $n\dot{P}_m(1 - P_m) = 0.9n$, where n is the number of genes in the genome.

The size of the mutation applied to a gene is the result of multiplying two distinct values drawn from a Gaussian distribution, $\mathcal{N}(0, \sqrt{\frac{\pi}{2}})$. A randomly drawn value from this distribution has an expected deviation from the mean, 0, of exactly 1, which is useful when the realised values during mutation are used in an adaptation process. The result of using the product of two values drawn from this distribution instead of just one is that the standard deviation of the deviation from the mean is larger. That is to say, the result, M_g, will vary more in size, with many small values and some very large values. While the variance of the absolute value of values drawn from this distribution is about 0.57, the variance of the absolute value of the product of two such values is about 1.465. However, the expected value remains at exactly 1. This is expressed in Section 3.2.1.

$$\begin{aligned} E(n_{g,1}n_{g,2}) = &\quad E(n_{g,1}) \cdot E(n_{g,2}) \\ 1 = &\quad 1 \cdot 1 \\ \\ Var(n_{g,1}n_{g,2}) = &\quad Var(n_{g,1})Var(n_{g,2}) + E[n_{g,1}]^2 Var(n_{g,2}) + E[n_{g,2}]^2 Var(n_{g,1}) \\ 1.465 = &\quad 0.57^2 + 2(1^2 \cdot 0.57) \end{aligned} \tag{3.2}$$

Adaptation

In general, for self-adaptive strategy parameters, the success of any specific value is determined by whether it produces viable offspring. This is to a certain extent up to chance, both because whether an individual will survive is quite dependent upon the state of the rest of the population and because strategy parameters themselves are not deterministic in their application. However, the adaptation that occurs when reacting to positive feedback, meaning the survival of an offspring, does not necessarily have to be random. By using the information about the outcomes from the probabilistic parts of the application of the strategy parameters, it is possible to make a somewhat educated guess about which way to change the respective parameter and by how much. In this way, the process of adaptation can become derandomised.

In the algorithm presented in this thesis, each individual has a strategy parameter for each gene in the genome. Each of these strategy parameters represents the standard deviation of the Gaussian distribution used for mutation of the respective gene. Equation 3.3 shows how these strategy parameters are adapted as they are passed on from parent to child. It is important to note that the adaptation of these strategy parameters occurs only if the value of the respective gene has changed from parent to child, that is to say, when there is something to adapt to. One could argue that no change is something to adapt to as well, but

since it is not the likelihood of change but rather the magnitude of change to the gene that is represented in the strategy parameter, we refrain from adapting the parameters when no change has occurred.

$$\sigma_{o,g} = \sigma_{p,g} + \frac{|X_{o,g} - X_{p,g}| - \sigma_{p,g}}{\tau} \tag{3.3}$$

$\sigma_{o/p,g}$: Standard deviation for the size of the distribution used for Gaussian mutation on gene g, for offspring (o) / parent (p).

$|X_{o,g} - X_{p,g}|$: The absolute difference between the parent's and the offspring's value for gene g. In other words, the mutation or change that happened to gene g during the creation of the offspring.

τ : User-defined parameter that determines the contribution of a single sample relative to the samples from previous generations. In other words, the memory of the parameter, or alternatively, the adaptation speed.

Because the expected absolute value for a mutation is $\sigma_{p,g}$, there is no implicit bias towards either decreasing or increasing the value as the number of mutations goes towards infinity. However, it is worth noting that the probability of getting a mutation value that is lower than $\sigma_{p,g}$ is about 0.66. Consequently, adaptations towards smaller mutation sizes are greater in number but smaller in size than adaptations towards bigger mutation sizes.

3.2.2 Recombination

In Section 2.1.3, we briefly discussed the most common recombination schemes for EA. We tested some of these schemes on our implementation and settled for N-point crossover between two individuals with 0.25 as the probability for crossover at any given point on the genome. Though recombination is not the focus of this thesis, the applied recombination scheme, if any, has a huge impact on the evolution of the mutation strategy parameters. A newly created individual will inherit the strategy parameters of only one of the parents used for recombination. Because these parameters adapt to any change in their respective gene, a successful recombination that resulted in a different value for one or several genes would influence the adaptation of the respective mutation strategy parameters. This will be further discussed in Sections 4.3.8 and 5.1.4.

3.2.3 Summary

To summarise, our approach can be viewed as a hybrid between DE and the derandomised adaptive mutation scheme introduced by Ostermeier et al. [1994]. Although the derandomised adaptation scheme introduced by Ostermeier et al. [1994] is similar to what we propose, there are some big differences between them. While their algorithm strives to mimic the contours of the fitness landscape around the entire population, ours is a slightly adapted version of the classical GA but gives each individual separate treatment with respect to the path it has travelled from its ancestors. Ostermeier et al. [1994] redefines the population each generation, while our implementation is more traditional.

The mutation scheme in DE uses the difference between the values of two arbitrary individuals within the population. This approach can be expected to have some of the same characteristics as the implementation we propose because of the utilisation of known good mutation vectors, which is illustrated in Figure 2.6, and because of the fact that crossover will affect the adaptive strategy parameters which will render them somewhat similar to mutation vectors in DE. However, a key difference is, again, that we separate each individual from the rest of the population and set the mutation based on what has worked for each individual in the past.

The main advantage of using a derandomised adaptation scheme is that the adaptation is done based on more reliable data. For classical self-adaptive strategy parameters, the parameters are subject to random mutation. This means that a child inherits its parent's strategy parameters but with a change to them that is determined by arbitrary mutation. Although GAs following the scheme presented here still have individuals inheriting their parents' strategy parameters, the alterations done to these parameters are determined not by random mutation but rather by the amount of change applied to their respective genes during the production of the new successful individual.

3.3 Selection Schemes

As we discussed in Section 2.1.4, diversity is the main reason for selecting individuals that are inferior in terms of fitness. Furthermore, our mutation operator is constructed in such a way that if finding a better solution is difficult, then the strategy that is most likely to succeed for the best individuals is to not explore at all. Thus, we suspect that the mutation parameters will converge to zero. This is a problem for most self-adaptive schemes. Consequently, we wanted to

experiment with a selection scheme that selects the individuals which are passed on to the next generation step where their contribution to diversity is rewarded.

Some selection schemes were discussed in Chapter 2, and we were torn between two schemes we found especially interesting. We implemented an adapted version of both of these and thoroughly tested them with different settings. Their architecture will be discussed in this section, along with a brief description of selection scheme used for selecting parents for mating.

3.3.1 Parent Selection

For parent selection we have used the classical roulette selection, where the probability of getting selected is given by a nonlinear ranking scheme based on the individual's fitness. This ranking scheme is adapted from Yu and Gen [2010] and is given by Equation 3.4, where $0 \leq \alpha \leq 1$ is the parameter that controls the selection pressure. The population is sorted by fitness score in descending order; the first individual is the one with the best fitness score. Equation 3.4 shows the probability of the i'th individual being selected, p_i. This results in the probability of the best individual being selected to be α, and the probability for other individuals being selected declines in a nonlinear manner after that.

$$p_i = \alpha(1 - \alpha)^i \tag{3.4}$$

3.3.2 Distance Measures

In order to give a measure of diversity contribution, a measure of distance between two individuals must be defined. The most minimalistic approach to this would be to simply define it as the Manhattan distance between the genomes of the two. However, there are several reasons why normalising the distance for each gene in the genome can bring about better approximations of how similar the two individuals are.

One reason for normalisation is that genes in the genome might have different ranges in which they take on their values. The obvious go-to solution for this is to normalise by dividing by the range that is predefined by the programmer. However, the range of possible values for the genes is not necessarily a good representation of the actual span of values that is realistic to find in good solutions to the problem. It is not unlikely that 90% of the predefined range for each variable provides practically useless solutions and that after just a few number

of generations, these intervals of useless values have been inspected for the last time. If so, it is probably better to treat the remaining 10% as though it were the entire range of values for the gene. The normalisation function as described by Equation set 3.5 accounts for this possibility by using the difference between the biggest and the smallest value of the gene that is currently represented in the population as the normalisation factor for that gene.

$$\begin{aligned} norm(g,P) &= \left\{ \begin{array}{ll} g_{max} - g_{min} & \text{if } g_{max} \neq g_{min} \\ \frac{range(g)}{10^5} & \text{otherwise} \end{array} \right\} \\ g_{min} &= \min_{i \in P} X_{i,g} \\ g_{max} &= \max_{i \in P} X_{i,g} \end{aligned} \tag{3.5}$$

$X_{i,g}$: The value of gene g, for individual i.

P : A population of individuals.

$norm(g,P)$: Normalisation factor for gene g, given population P.

In addition to partially taking into account potential useless intervals of gene values, this normalisation method also rewards diversity for each gene based on the current range of values represented for this gene in the population. That is to say, it rewards diversity where there is little of it, which is another reason for normalising when trying to approximate contribution to diversity. We can now define the normalised distance between two individuals as follows (Equation 3.6):

$$Dist(i,j,P) = \sum_{g \in G} \frac{|X_{i,g} - X_{j,g}|}{norm(g,P)} \tag{3.6}$$

G : The number of genes in the genome.

3.3.3 K-Nearest Neighbours

The most similar scheme to the one proposed next is perhaps deterministic crowding, as presented by Mengshoel and Goldberg [1999]. In deterministic crowding, the individuals are grouped in such a way as to maximise the similarity of individuals within each group, and the individuals with the highest fitness score in each

group are chosen for survival. A difficulty with the implementation of this idea is that the problem of finding the groupings that maximise similarities within the groups is itself an NP-hard problem. Consequently, a good approximation is the closest one can get, and the question of what algorithm to use is raised.

Inspired by these simple approaches, we implemented a selection scheme where each selected individual removes its k-nearest neighbours from the candidate pool before a new individual is selected. The pseudocode for this selection process is given in Algorithm 1.

Data: *Population*, *GenerationSize*, *SurvivalRate*
Result: *Survivors*

$SurvivorCount \leftarrow GenerationSize \times SurvivalRate$
$K \leftarrow \lfloor \frac{|Population| - SurvivorCount}{SurvivorCount} \rfloor$
$Candidates \leftarrow Population$
$Survivors \leftarrow \emptyset$

while $|Survivors| < SurvivorCount$ **do**
  $Best \leftarrow$ Individual in candidates with the best fitness score
  $Survivors \leftarrow Survivors + Best$
  $Candidates \leftarrow Candidates - Best$
  for *i=1 to K* **do**
    $NearestNeighbour \leftarrow$ individual in *Candidates* that is closest to *Best*
    $Candidates \leftarrow Candidates - NearestNeighbour$
  end
end

return *Survivors*

Algorithm 1: Selection Scheme: K-Nearest Neighbours. Note that the measuring of distance follows Equations 3.6 and 3.5, with the set of Survivors as the population, P in 3.5. Note also that this implementation is suboptimal in terms of execution speed. It is used here to simplify the interpretation of the result.

This approach is quite similar to deterministic crowding, differing mainly in the incremental formation of crowds. Each crowd is centred on the best individual in the remaining candidates, and the only concern is the distance to this particular individual. We have not found any good reasons for taking into account the distance between individuals that will not be included in the set of survivors. Hence, it makes sense to centre the groupings on specific individuals after they

have been chosen for survival and minimise the sum of distances to the selected individual within the groupings.

3.3.4 Contribution to Diversity as an Objective

Inspired by de Jong et al. [2001], we wanted to employ a selection scheme to GA which uses diversity measures directly as an objective. Its architecture and the motivation behind the choices made throughout its development are described in this section.

Measures of Contribution to Diversity

The first question to address was how to quantify an individual's contribution to diversity. Under the construction of the algorithm, we were especially torn between two measures: distance to closest neighbour (Equation 3.7) and average distance to all selected individuals (Equation 3.8).

$$CD(c, Survivors) = \min_{s \in Survivors} Dist(c, s) \tag{3.7}$$

$$CD(c, Survivors) = \sum_{s \in Survivors} Dist(c, s) \tag{3.8}$$

$CD(c, Survivors)$: Candidate c's contribution to diversity, given already-selected survivors.

$Survivors$: Individuals that have already been selected for survival.

c : Arbitrary candidate for survival selection.

de Jong et al. [2001] used the average distance to all other individuals as a measure of contribution to diversity. Lozano et al. [2008], on the other hand, used the distance to the closest neighbour as the measure of an individual's contribution to diversity. Their steady-state implementation uses a replacement scheme where each new individual eliminates the individual with the lowest contribution to diversity among the individuals with a worse fitness score than itself, given that the lowest-scoring individual has a lower contribution to diversity than the new one. If not, the new individual replaces the worst individual in the population, given that it scored better than the worst individual in terms of fitness.

From Equations 3.7 and 3.8, it follows that the selection process is incremental. For each individual that is selected for survival, the remaining individuals' contributions to diversity must be re-evaluated, taking the newly selected individual into account. This makes for a time-consuming selection process as the number of individuals rises. Its run time is $\mathcal{O}\left(\frac{(n-1)!}{(n-a)!}\right)$, where n is the number of individuals and a is the number of individuals in the final selection. Consequently, this scheme is rational to apply only when fitness testing is far more time consuming than the GA.

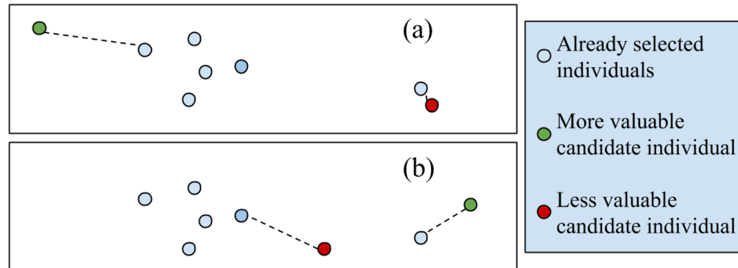


Figure 3.1: Illustrates two scenarios where shortest distance and average distance will lead to different decisions, when selection is done considering the contribution to diversity.

Figure 3.1 illustrates two scenarios where shortest distance and average distance will lead to different decisions. In the first scenario, (a), the individual on the left clearly contributes more to the diversity of the population than the candidate on the right. However, choosing the candidate on the left partially inhibits further exploration around the neighbour of the candidate on the right. However, in the second scenario, (b), the opposite seems to be the case. We consider the candidate on the right to contribute more since it is almost as far away from its nearest neighbour as the candidate on the left, and since its nearest neighbour is an exploring candidate itself.

Using shortest distance, there are only two ways in which an exploring individual can produce successful offspring. The first is to find a solution that replaces itself. The second is to make big leaps in solution space to get a high contribution to diversity even when its parent is still alive. That being said, such examples might not make much sense in a solution space with 100 dimensions, or even 10 for that matter.

Weighing Contribution to Diversity against Fitness

In Section 3.3.2, we discussed how distance between two individuals can benefit from being normalised with respect to each gene and the different values represented within the population. In the case of weighing CD against fitness, normalisation is at least as important. Difference in magnitude for these measures is likely to be huge. The most natural solution in this case is the equivalent of the normalisation factor expressed by Equation set 3.5. In other words, the size of the range of values present in the set of remaining candidates. However, in the case of contribution to diversity, we know the minimum possible value, which is 0, and we therefore substitute g_{min} with 0. The multi-objective score for an individual is then simply calculated by adding them together with CD multiplied by a user defined weight, as expressed by Equation 3.9.

$$Score(c, normFitness, normCD) = \frac{Fitness(c)}{normFitness} + \alpha \frac{CD(c)}{normCD} \tag{3.9}$$

The weight, α, will have a significant impact on the behaviour of the algorithm and is therefore subject to fine-tuning. Algorithm 2 shows the pseudocode for this selection scheme.

3.4 Implementation

This section presents an overview of the implementation of the GA and the system used to test and display various aspects of the GAs. We will explain how the huge amount of data will be managed and how this data will be used for different types of visualisations in the GUI.

3.4.1 Genetic Algorithm

In order to compare multiple versions of our own algorithm both to one another and to other well-known algorithms, we have implemented an easy mouldable core to which it is easy to add any number of different operators and selection schemes. The core of the GA is the glue which holds the different modules together.

Figure 3.2 is a simple illustration of what this core looks like. From the figure we can see the core needs a ParentSelector, a set of Operators, a Phenotyper, a Tester and a SurvivalSelector. All these components are interfaces consisting of

```
Data: Population, GenerationSize, SurvivalRate, Weight
Result: Survivors

SurvivorCount ← GenerationSize × SurvivalRate
Candidates ← Population
Survivors ← ∅
s ← Individual in Candidates with best fitness score
for c in Candidates do
|  c.CD ← +∞
end

while |Survivors| < SurvivorCount do
    for c in Candidates do
    |  c.CD ← Min(c.CD, Dist(c, s))
    end

    normFritness ← Range in fitness among Candidates
    normCD ← Range in CD amond Candidates

    bestScore ← −∞
    for c in Candidates do
        score ← Score(c, normFitness, normCD)
        if score > bestScore then
            bestScore ← score
            s ← c
        end
    end

    Survivors ← Survivors + s
    Candidates ← Candidates − s
end

return Survivors
```

Algorithm 2: Selection Scheme: Diversity as Objective. Note that the measuring of distance follows Equations 3.6 and 3.5, with the set of Survivors as the population, P in 3.5. Note also that this implementation is suboptimal in terms of execution speed. It is used here to simplify the interpretation of the result.

one or two methods each. In Figure 3.2 there are three operators; however, any number of operators is possible. We found the unlimited number of operators to

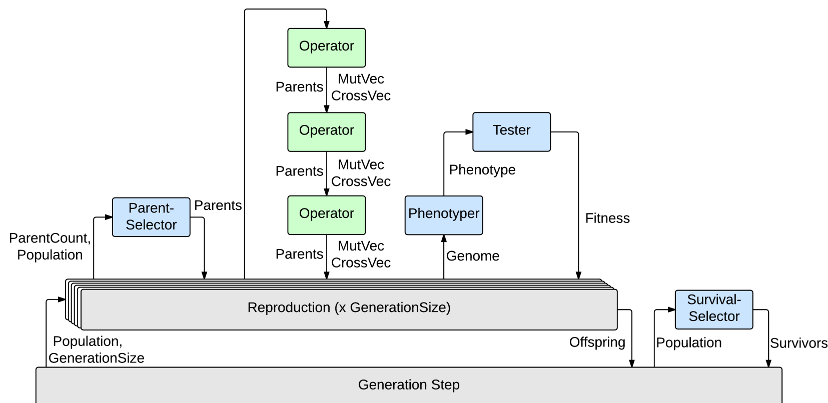


Figure 3.2: A state diagram of the GA implementation giving an overview of the structure and illustrating how the modules interact during a generation step of the GA.

be quite useful because it made it possible to test many different combinations of operators without having to change the code in between each test. Figure 3.4b in Section 3.4.2 shows the benefit of this architecture during testing.

The operator interface contains a single method: ApplyOperator, which takes as parameters the parents, a mutation vector and a crossover vector. Both the mutation vector and the crossover vector are as long as the genome and start out containing only zeros. Each operator changes these vectors as they are applied, and finally the vectors are used along with the parent genomes to create the genome of the offspring, as depicted by Figure 3.3.

Each number in the final crossover vector dictates from which parent the gene should be taken, with the number being the parent's index in the parent array. A "0" on the i'th place in this vector means that the i'th gene of the offspring should be taken from the first parent in the parent array. After the child genome has been put together from values in the parents' genomes, the mutation vector is simply added to the genome vector.

3.4.2 Graphical User Interface and Data Management

As mentioned in Section 1.3, we wanted to have a system for data management and visualisations of the data from different runs of the EA implementation.

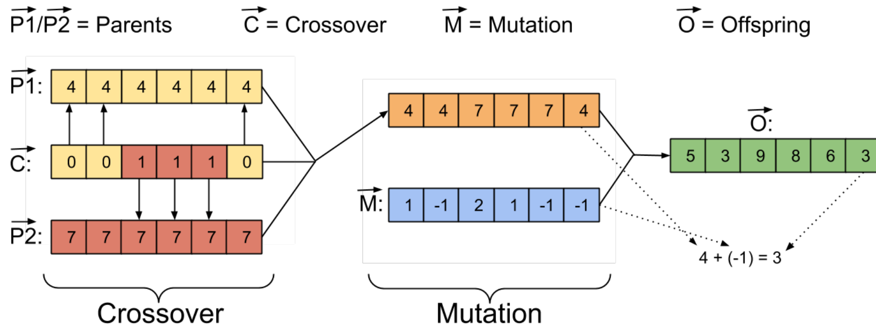
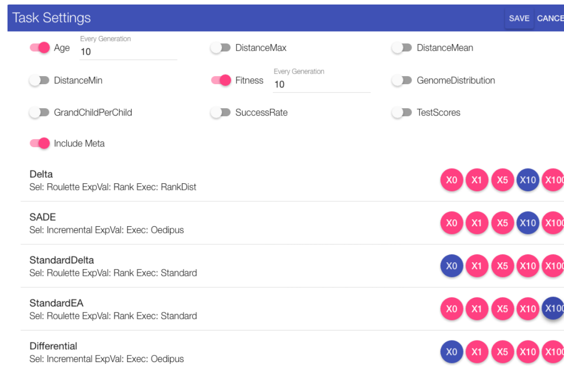


Figure 3.3: Illustration of how the mutation vector, the crossover vector and the parents' genomes are used to construct the offspring.

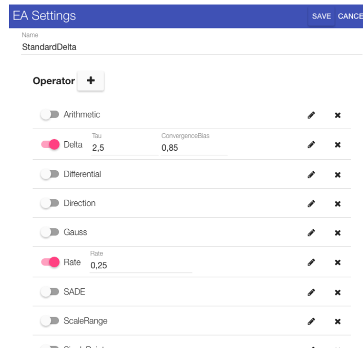
This system was achieved by implementing a web server. A web server provided us with a simple way of creating the GUI through HTML and CSS and some JavaScript. By using an existing web framework, we could easily connect the server to a database for storing, managing and serving the vast amounts of data generated by the GA. Storing all this data made it possible to generate tables, graphs and different types of visualisations from different runs of the GA.

The code for the web server is completely separated from the GA implementation mentioned in Section 3.4.1. The only integration the GA implementation needed was a way to receive tasks from the web server. This was done by connecting the GA implementation to TCP sockets hosted by the server. The task which is assigned is effortlessly created in the GUI, as shown in Figure 3.4b. Here we have taken advantage of the highly mouldable core of our GA implementation by using check boxes to enable the different modules discussed in the previous section. Here we can also define the modules' respective parameters.

We also implemented a method for queueing multiple tasks with different module settings for the GA; this is shown in Figure 3.4a. As seen here there are some check boxes to specify what kind of data should be stored from the run in the database and how often. For example, in Figure 3.4a, the checkboxes for Age and Fitness are checked and will be included every 10 generations from the GA. Both having this queue of tasks and connecting GAs through TCP sockets enabled us to take advantage of multiple computers running the GA implementation by connecting them to the server. This made it less time consuming to do the required testing and experimenting with the GA. We have created a video showing how the process of creating a new setup for an GA, and also how a task can be queued: http://folk.ntnu.no/thafveli/createEA



(a) Illustrates how tasks can be queued in the GUI



(b) Modifying the core of GA through the GUI

Figure 3.4: Pictures from the GUI

3.4.3 Visualisation

For visualisation we have implemented multiple methods of inspecting the GA implementation. In this section, we will discuss the most important visualisation tools used. A description of other visualisation tools created is included in appendix A.2, a set of videos from the GUI can also be found in the appendix A.1.

The main view of the GUI is a very basic list of all the runs of the GA. Here we have created methods enabling us to easily separate different runs with different settings and see how well they performed. We also implemented a graph tool where we can view the parameters for each generation which were selected when queueing the tasks. The graph tool will be used to present some of the results

in the next chapter. Here we will also discuss the different types of parameters which can be inspected.

Advanced Visualisation

For more advanced inspection of how the GA implementation behaves in the solution space, a method enabling us to inspect the genomes of the best individual and that individual's ancestors was implemented. This visualisation shows how the genome moves in the solution space for each ancestor of the best individual. It works by mapping the genome of each individual to a two-dimensional plane where the genomes are grouped together in pairs. The odd-indexed genes are then given a position on the x-axis of the plane, and the even-indexed genes get a position on the y-axis. For example, a real-valued genome $g = [0.1, 0.2, 0.4, 0.5]$ would be mapped as two points in the two-dimensional plan, placed at $[0.1, 0.2]$ and $[0.4, 0.5]$.

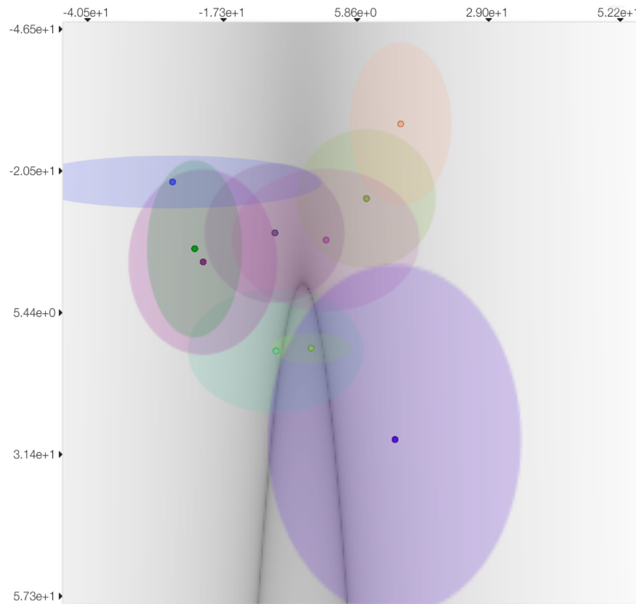


Figure 3.5: Advanced visualisation through the GUI, showing how the best individual's and its ancestors' genomes move in solution space.

In Figure 3.5, a screenshot taken from the GUI shows how this visualisation looks. The figure shows the genome of one individual trying to optimise the

20-dimensional Rosenbrock function. Therefore, it is possible to see 10 small coloured dots, one dot showing the placement of a pair of genes. The visualisation also shows a big circle around the dots; this symbolises the mutation step size, σ, for that particular gene pair. The mutation step-size circle is visible only for GAs using the adaptive mutation scheme presented in Section 3.2.1.

In the background of the visualisation, we render a black-and-white heat plot of the benchmark function, where black symbolises minimums of the function. Figure 3.5 clearly shows the distinctive valley-shaped contours of the Rosenbrock function, which also can be seen in the 3D heat plot of Rosenbrock in Figure 4.1b in the next chapter.

3.5 Technology

In this section, we will summarise the different technologies used for the implementation.

3.5.1 Go Programming Language

For the GA implementation, we wanted to use a language that could easily provide us with a concurrent environment so we could exploit the parallel architecture of modern computers. We therefore chose Go Programming Language.

Go Programming Language (Golang) [Pike, 2009] provides a concurrency through simple primitives. This makes it easy to construct streaming data pipelines that make efficient use of both the I/O and all CPUs.

3.5.2 Sails.js Server

As mentioned earlier, we wanted to separate the EA implementation from the GUI and storage. We decided to use the Sails.js framework, which runs on Node.js, for the server. The reasoning behind this was that we were familiar with this framework, it is simple to set up, it is flexible and it provides many powerful extensions for data storage.

3.5.3 MongoDB

To manage the vast amounts of data sent from each generation of the EA, we utilised MongoDB, which is a schemaless database providing fast storage and search. The schemaless property was an important criterion because it made it easy and less time consuming to create and change schemes for the data stored in the database.

Chapter 4

Experiments and Results

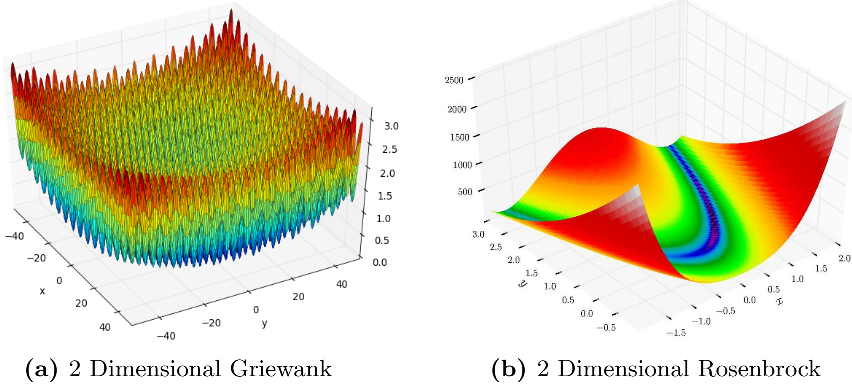
In this chapter, we will present the experimental setup and the results obtained from various tests of the architecture suggested in chapter 3. In Section 4.1, we present the different tests that have been conducted and the setup for each of them. In Section 4.2, the setup of our implementations of the GA will be presented, along with three other GA implementations and their respective setup. Finally, in Section 4.3, the results achieved for each of the GA implementations will be presented.

4.1 Test Setup

To address the goals and research questions defined in 1.2, we have implemented 21 common mathematical benchmark functions for optimisation problems, as well as two more-practical problems: pole-balancing problem and robot arm. In this section, we will discuss the selected tests and their setup.

4.1.1 Test Functions

An important criterion for the test functions is the ability to compare results to that of similar algorithms in related work. For this reason, we have implemented 21 common test functions for optimisation. These test functions provide insight into the general performance of different algorithms, as well as into what types of functions a particular algorithm is more or less befitting. Furthermore, because



(a) 2 Dimensional Griewank **(b)** 2 Dimensional Rosenbrock

Figure 4.1: Landscape of mathematical benchmark functions

the fitness landscape of each one these functions is known, they are also quite useful for analytical purposes.

The most widely used optimisation problems for GA are multimodal functions like Ackley, Griewank, Levy and Rastrigin. Although these functions have a general decrease towards their global minimum, they are difficult because they do not decrease monotonically towards the minimum. This can be seen in the figure presenting the fitness landscape of Griewank, Figure 4.1a. The global optimum is easily spotted, and there is a general decrease of function value towards it, but there are no monotonically decreasing paths to it. Functions like Rosenbrock, Beale, Booths and Matyas, on the other hand, do have paths towards the minimum on which the function value decreases monotonically. This can be seen in the figure presenting the fitness landscape of Rosenbrock, Figure 4.1b. However, their challenge is to coordinate the change in one gene to the change in the other gene that is used in the function. Last but not least, there are functions that have quite specific challenges related to them. Among these are Easom, Step and Bunkin. For instance, Easom's fitness landscape is flat everywhere but for a small area around π in which the function value drops to zero.

In Tables 4.1 and 4.2, we have listed the selected test functions, all of which were obtained from Jamil and Yang [2013]. We have split the table into some categories to separate the different types of landscape they have.

Many Local Minima				
#	**Name**	**Function**	**D**	**Range**
$f1$	Ackley	$f(x) = 20 + e - 20\exp\left(-0.2\sqrt{\frac{1}{n}\sum_{i=1}^{D} x_i^2}\right)$	100	$-5 \leq x, y \leq 5$
$f2$	Ackley 2D	$f(x,y) = -20\exp\left(-0.2\sqrt{0.5\left(x^2+y^2\right)}\right) - \exp\left(0.5\left(\cos\left(2\pi x\right) + \cos\left(2\pi y\right)\right)\right) + e + 20$	100	$-5 \leq x, y \leq 5$
$f3$	Bukin N.6	$f(x,y) = 100\sqrt{\left\lvert y - 0.01x^2\right\rvert} + 0.01\left\lvert x + 10\right\rvert.$	50	$-15 \leq x \leq -5$ $-3 \leq y \leq 3$
$f4$	Cross-in-tray	$f(x,y) = -0.0001\left(\left\lvert \sin\left(x\right)\sin\left(y\right)\exp\left(\left\lvert 100 - \frac{\sqrt{x^2+y^2}}{\pi}\right\rvert\right)\right\rvert + 1\right)^{0.1}$	100	$-10 \leq x, y \leq 10$
$f5$	Eggholder	$f(x,y) = -\left(y+47\right)\sin\left(\sqrt{\left\lvert y + \frac{x}{2} + 47\right\rvert}\right) - x\sin\left(\sqrt{\left\lvert x - \left(y+47\right)\right\rvert}\right)$	10	$-512 \leq x, y \leq 512$
$f6$	Griewank	$f(\vec{x}) = \frac{1}{4000}\sum_{i=1}^{D}(x_i - 100)^2 - \prod_{i=1}^{D} cos\left(\frac{x_i - 100}{\sqrt{i}}\right)$	100	$-600 \leq x \leq 600$
$f7$	Hölder table	$f(x,y) = -\left\lvert \sin\left(x\right)\cos\left(y\right)\exp\left(\left\lvert 1 - \frac{\sqrt{x^2+y^2}}{\pi}\right\rvert\right)\right\rvert$	100	$-10 \leq x, y \leq 10$
$f8$	Levy	$f(x) = sin^2(\pi w_1) + \sum_{i=1}^{d-1}(w_i - 1)^2(1 + 10sin^2(\pi w_i + 1)) + (w_d - 1)^2(1 + sin^2(2\pi w_d))$ $w_i = 1 + \frac{x_i - 1}{4}$	100	$-10 \leq x \leq 10$
$f9$	Rastrigin	$f(\vec{x}) = \sum_{i=1}^{D}(x_i^2 - 10\cos(2\pi x_i) + 10)$	100	$-5.12 \leq x \leq 5.12$
$f10$	Schaffer N. 2	$f(x,y) = 0.5 + \frac{\sin^2\left(x^2 - y^2\right) - 0.5}{\left(1 + 0.001\left(x^2+y^2\right)\right)^2}$	10	$-100 \leq x, y \leq 100$

Bowl-Shaped				
#	**Name**	**Function**	**D**	**Range**
$f11$	Sphere	$f(\boldsymbol{x}) = \sum_{i=1}^{n} x_i^2$	100	$-\infty \leq x_i \leq \infty$, $1 \leq i \leq n$

Table 4.1: Definition of the benchmark functions, and the range of the parameters used for testing of the implementation. Column D defines the dimensionality used when testing, if nothing else is specified

#	Name	Function	D	Range
Plate-Shaped				
$f12$	Booth's	$f(x,y) = (x+2y-7)^2 + (2x+y-5)^2$	100	$-10 \leq x, y \leq 10$
$f13$	Matyas	$f(x,y) = 0.26\left(x^2+y^2\right) - 0.48xy$	100	$-10 \leq x, y \leq 10$
$f14$	McCormick	$f(x,y) = \sin(x+y) + (x-y)^2 - 1.5x + 2.5y + 1$	50	$-1.5 \leq x \leq 4$, $-3 \leq y \leq 4$
Valley-Shaped				
$f15$	Three-hump	$f(x,y) = 2x^2 - 1.05x^4 + \frac{x^6}{6} + xy + y^2$	100	$-5 \leq x, y \leq 5$
$f16$	Rosenbrock	$f(\boldsymbol{x}) = \sum_{i=1}^{n-1}\left[100\left(x_{i+1} - x_i^2\right)^2 + (x_i - 1)^2\right]$	20	$-\infty \leq x_i \leq \infty$, $1 \leq i \leq n$
Steep Ridges/Drops				
$f17$	Easom	$f(x,y) = -\cos(x)\cos(y)\exp\left(-\left((x-\pi)^2 + (y-\pi)^2\right)\right)$	100	$-100 \leq x, y \leq 100$
Other				
$f18$	Beale's	$f(x,y) = (1.5 - x + xy)^2 + \left(2.25 - x + xy^2\right)^2 + \left(2.625 - x + xy^3\right)^2$	40	$-4.5 \leq x, y \leq 4.5$
$f19$	Goldstein–Price	$f(x,y) = \left(1 + (x+y+1)^2\left(19 - 14x + 3x^2 - 14y + 6xy + 3y^2\right)\right) \left(30 + (2x-3y)^2\left(18 - 32x + 12x^2 + 48y - 36xy + 27y^2\right)\right)$	30	$-2 \leq x, y \leq 2$
$f20$	Step	$\sum_{i=1}^{D} \lfloor \lvert x_i \rvert \rfloor$	100	$-100 \leq x \leq 100$
$f21$	Styblinski–Tang	$f(\boldsymbol{x}) = \frac{\sum_{i=1}^{n} x_i^4 - 16x_i^2 + 5x_i}{2}$	100	$-5 \leq x_i \leq 5$ $1 \leq i \leq n$

Table 4.2: Definition of the benchmark functions and the range of the parameters used for testing of the implementation. Column D defines the dimensionality used when testing, if nothing else is specified

Dimensionality

One advantage with the use of mathematical test functions is that the dimensionality of the problems are easily changed. We will primarily use 100 dimensions for the benchmark functions; however, for some of the test functions, we have decreased the number of dimensions. The reason for this is that we wanted to bring about the most interesting results. Consequently, whenever none of the implementations were able to get close to the global optimum, we decreased the number of dimensions until at least one of them did. This was done to retrieve more interesting data from the test functions. Column D of Tables 4.1 and 4.2 defines the dimensionality that was used for the runs of the respective function, but we might specify otherwise for specific runs.

Increasing and decreasing the difficulty is not the only benefit of being able to change the number of dimensions to the problem. One can also test the robustness of an algorithm on different dimensionalities in general. Some algorithms are more capable of solving problems of different dimensionality than others, and some algorithms have input parameters that can be changed to adapt to different types of problems and different numbers of dimensions.

Rotation of Solution Spaces

A general disadvantage of using the mathematical test functions defined in Tables 4.1 and 4.2 is that most of them are separable for each gene, or for each pair of genes. All values for a specific gene, or pair of genes, have a specific contribution to the fitness score regardless of the state of the rest of the genome. In other words, it is possible to solve the problem for one or two dimensions at a time and find the global optimum this way. However, by rotating the solution space, this possibility is lost, and the contribution to fitness that results from changing the value of a gene is dependent upon the values of all the other genes as well. Figure 4.2 is a simple illustration of how changing the value of a gene affects more than one parameter in the final rotated solution and consequently how the optimal value of either of the two genes change with the value of the other gene. With randomised rotation, 100 dimensions and many local optima along each axis, the dependencies between genes at different points in the solution space would, of course, be much more complex.

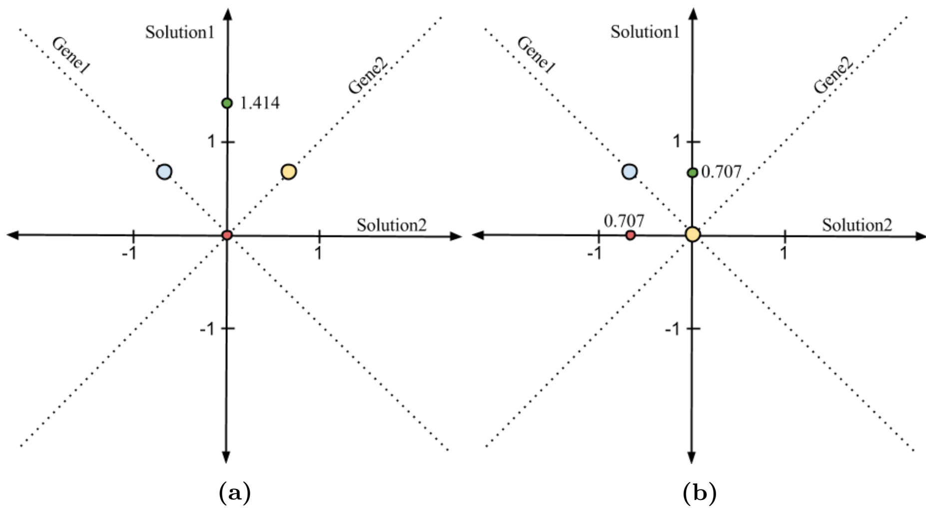


Figure 4.2: Illustration of how rotation of the solution space can turn a separable problem into a non-separable problem. ○ and ○ represent the values of Gene1 and Gene2, while ● and ● represent the corresponding values applied in the final solution after rotation. Simply imagining the problem is to minimize the sum of each solution value is sufficient to see that the effect of a change to Gene2 is dependent upon the value of Gene1, and vice versa. Changing the value of Gene2 from zero to one bares no effect on the fitness of the solution, as $0.707 + 0.707 = 1.414$. In a more complicated scenario with local optimums along each axis, the dependency is even stronger.

To achieve the equivalent of a rotated solution space, the vector solution of an individual is rotated with a rotation matrix which was created under the initialisation of the search. The rotation matrix rotates a vector of 1s to a vector whose values are taken randomly from the uniform distribution between -1 and 1, $\mathcal{U}(-1, 1)$ and then normalised to be of the same length as the vector of 1's.

4.1.2 Pole-Balancing Problem

Pole balancing is a problem with a more practical grounding than the mathematical optimisation functions, and it comes from the field of ANN and control theory [Brownlee, 2005]. The problem consists of a cart and an inverted pendulum (pole). While the cart is restricted to move on the horizontal axis, the pole is connected to the top of the cart and pivots freely around its point of connection. The goal is to balance the pole for as long as possible by applying a horizontal force to the cart.

The control system of the pole-balancing problem is a closed feedback loop with four input parameters and two output parameters. The four real-valued parameters, θ, $\dot{\theta}$, x and $\dot{x}$, make up the input. θ is the angle of the pole, $\dot{\theta}$ is the angular velocity of pole, x is the position of the cart and $\dot{x}$ is the velocity of the cart. The controller outputs two parameters, o_1 and o_2, which are used as shown in Section 4.1.2 to determine the force, F_t, applied to the cart. sgn is a function that returns -1 if its input is less than zero, 1 if its input is greater than zero and 0 if its input is zero. The resulting force is aligned with the cart and affects its movement along the x-axis.

$$\begin{aligned} \Delta_o = &\quad o_1 - o_2 \\ F_t = &\left\{ \begin{array}{ll} F_{max} * sgn(\Delta_o) & \text{if } |\Delta_o| > 0.2 \\ 0 & \text{otherwise} \end{array} \right\} \end{aligned} \tag{4.1}$$

Symbol	**Definition**	**Value**
g	Gravity	$9.81m/s^2$
d	Test Duration	$600s$
τ	Time step size	$0.02s \Rightarrow 50$Hz
h	Track limit	$2.4m$
r	Pole failure angle	12 degrees
m_c	Mass of cart	$1kg$
m_p	Mass of pole	$0.1kg$
l_p	Pole length	$0.5m$
F_{max}	Max force	$1N$

Table 4.3: Pole balancing variables

The cart's position and velocity, as well as the angle and the angular velocity of the pole, are updated using the discrete time function defined by Section 4.1.2, where t defines the current time step and τ is the size of the time steps. The discrete time function requires both the acceleration of the cart and the angular acceleration of the pole; these are calculated following Section 4.1.2. The system also requires a set of constants, which are presented in Table 4.3.

$$\begin{aligned} x_{t+1} =& \quad x_t + \tau\dot{x}_t \\ \dot{x}_{t+1} =& \quad \dot{x}_t + \tau\ddot{x}_t \\ \theta_{t+1} =& \quad \theta_t + \tau\dot{\theta}_t \\ \dot{\theta}_{t+1} =& \quad \dot{\theta}_t + \tau\ddot{\theta}_t \end{aligned} \tag{4.2}$$

$$\begin{aligned} \ddot{\theta}_t =& \quad \frac{g\sin\theta_t + \cos\theta_t\left(\frac{-F_t - m_p l\dot{\theta}_t^2 \sin\theta_t}{m_c + m_p}\right)}{l\left(\frac{4}{3} - \frac{m_p\cos^2\theta_t}{m_c+m_p}\right)} \\ \ddot{x}_t =& \quad \frac{F_t + m_p l\left\lfloor\dot{\theta}_t^2\sin\theta_t - \ddot{\theta}_t\cos\theta_t\right\rfloor}{m_c + m_p} \end{aligned} \tag{4.3}$$

The fitness score of the pole-balancing problem is set to the number of times the actor fails to balance the pole during the test. A test lasts for 10 minutes, but if an actor fails 200 times before the time has run out, the number of fails will be estimated based on the performance so far. Whenever the angle of a pole is more than θ, determined by the number of times the pole's angle $\theta > r$ during the duration, d, of the test, the test is reset with a new random pole angle. If the pole falls over more than a certain number of times, the test is terminated, and the individual is assigned the projected number of failures it would have gotten if the test had completed its duration. We also reset the test every 60 seconds, giving the pole a new random start angle, to make sure the controller was able to balance the pole in different situations.

To increase the difficulty, we have also conducted tests on the double pole-balancing problem. Double pole balancing is a quite simple extension to the balancing pole problem. In this extension, a second pole is to be balanced along with the first. Although the second pole follows the same set of rules as the first, it is shorter and will therefore fall faster than the first. A natural consequence of adding the second pole is the introduction of two new parameters to the feedback loop, θ_2, the angle, and $\dot{\theta}_2$, the angular velocity, for the second pole. Furthermore, two static variables are required: the mass for the second pole, $m_{p2} = 0.04kg$, and length of the second pole, $l_{p2} = 0.2m$. We also had to increase F_{max} to be $2N$ because of the extra mass added by the second pole.

Artificial Neural Networks

We used an evolving artificial neural network (ANN) as the controller of the pole-balancing feedback loop. To give a short summary of what an ANN is, it

is a system inspired by biological nervous systems. ANNs consist of a set of interconnected nodes, called neurons, which work in parallel to solve a specific problem. Each neuron has a set of inputs and outputs, called connections. The output from a neuron to a connection is defined by the connection's numeric weight and the neuron's activation function. This activation function produces a numerical output, usually between 0 and 1, based on the input received. We have used the Sigmodial activation function defined by Equation set 4.4.

$$\begin{aligned} SI_t = & \sum_{i=1}^{|IN|} w_{N_i} N_{i,t-1} \\ y_t = & y_{t-1} + \frac{SI - y_{t-1} + Bias}{Tau} \\ \sigma(y) = & = \frac{1}{1+e^{-\beta y}} \end{aligned} \tag{4.4}$$

We use a feed-forward ANN with layer base structure; this means the ANN is divided into multiple layers of neurons, where each neuron is connected to all neurons on the next layer. For the pole-balancing problem, only two layers were required to get a perfect score on the test, the input layer and the output layer.

Genetic Representation

For each neuron that is not an input, three parameters are evolved: $Bias$ in the range [-3, 3], Tau in the range [1, 5] and β in the range [1, 10]. In addition, and most importantly, the weight of each connection, w_{N_i}, is evolved in the range [-1, 1].

4.1.3 Robot Arm Function

Robot arm is a more practical problem commonly used in neural network literature [An and Owen, 2001]. It has been adapted to an optimisation problem which consists of finding angles for joints and lengths for arm segments. The goal is then to get the end of the arm is as close as possible to the target position; this be seen in Figure 4.3. Each joint can take on any angle between 0 and 2π, and each arm segment can take on a length between 0 and 1. Table 4.4 displays the input parameters, which are directly mapped from the genome. The target position is generated at the beginning of a search, as expressed by the set of Equations 4.5.

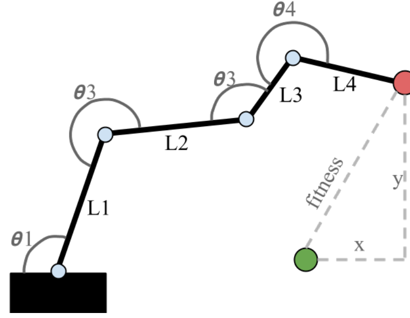


Figure 4.3: Robot arm with four segments. θ defines the angle between two joints, while L gives the length of a segment. The goal is to change the θs and Ls so ● is moved to ●

Symbol	Definition	Value
θ_i	Angle of the i'th joint	$[0, 2\pi]$
L_i	Lenght of the i'th arm segment	$[0, 1]$

Table 4.4: Input Parameters for Robot Arm Function

$$
\begin{aligned}
L_t &= 1 + r_1(S-1) \\
\theta_t &= r_2\pi \\
x_t &= L_t Cos(\theta_t) \\
y_t &= L_t Sin(\theta_t)
\end{aligned}
\tag{4.5}
$$

r_1 and r_2 are taken randomly from the uniform distribution between 0 and 1. The fitness score is simply the distance between the end of the arm and the target, as shown in Figure 4.3. The Equation set 4.6 shows how the end position, x and y, of the robot arm is calculated, as well as the final fitness score.

$$
\begin{aligned}
x &= \sum_{i=1}^{n} L_i Cos\left(\sum_{j=1}^{i} \theta_i\right) \\
y &= \sum_{i=1}^{n} L_i Sin\left(\sum_{j=1}^{i} \theta_i\right) \\
fitness &= \sqrt{(x-x_t)^2 + (y-y_t)^2}
\end{aligned}
\tag{4.6}
$$

4.2 Experimental Setup

We have selected a set of algorithms from the related work to implement and compare our implementation with. The deciding factors when choosing the algorithms to compare were their relation to the work of this thesis and also the results presented in the paper. In this section, we present the selected algorithms, the reason why we selected them and the parameter values used when testing them.

To get a fair comparison of the different algorithm implementations, we have set the number of offspring per generation to be the same for all of them. This implies that all algorithm implementations do the same number of fitness evaluations each generation, which enables us to see the robustness of the different GA implementations. Each of the implementations has a population size of 100 individuals at the start of each generation step and performs 100 fitness evaluations to test new offspring for each generation step.

Adaptive Mutation Only (AM)

In the GA setting marked AM, we use the adaptive mutation scheme described in Section 3.2, alongside somewhat regular GA settings. For recombination we use N-point crossover, and for parent selection we use roulette selection with nonlinear ranking, given by the Section 3.3.1, with $\alpha = 0.02$.

During survival selection, we simply selected the best individuals with no exceptions. This implies 100% elitism. The number of survivors is set to 100% of the number of individuals created each generation, which corresponds to 100 survivors when the generation size is 100. In other words, of the 100 survivors of the previous generation and the 100 newly created offspring, 100 will survive to make up the parent pool in the next generation. For the rate of adaptation, we found $\tau = 1.5$ to give the best results, although up to 3 seems to give somewhat similar results.

Adaptive Mutation using K-Nearest (AM-KN and AM-KN*)

The difference between AM and AM-KN is the selection scheme used during the selection of survivors each generation. While AM uses the simplest selection scheme possible, AM-KN makes an effort to maintain diversity within the population by using the selection scheme discussed in Section 3.3.3. As with AM, AM-KN also has 100 survivors, which is the same as the number of offspring created each generation.

We have also created a different setup for AM-KN as an effort to push the speed of convergence to a level comparable with that of AM; we will refer to this as AM-KN*. This setup uses the same parameters as AM-KN except for the parameter controlling the selection pressure during parent selection, α in Section 3.3.1, which is set to 0.05. We also increased τ to 2.5 for the mutation operator.

Differential Evolution (DE)

Differential evolution (DE), which was discussed Section 2.1.6, is one of the algorithms we chose to compare ourselves with. This is a very interesting implementation because of the mutation vector it utilises, which shares some similarities with our implementation; this will be investigated more closely in Section 5.1. The results presented on the mathematical benchmarks in Storn and Price [1997] are also quite good.

DE requires only two parameters to be defined: the amplification factor of the differential variation, F, and the crossover rate, CR. We used the recommended initial values for these, which are

$F = 0.5$
$CR = 0.9$

Self-Adaptive Differential Evolution (SA-DE)

In Section 2.2.2, we presented the self-adaptive differential evolutionary (SA-DE) algorithm. This is a very interesting implementation because of its simplicity, and the results it achieved on the mathematical optimisation problems are very good. Overall, SA-DE is the most impressive algorithm we have seen in our study of the field in terms of results relating to both robustness, speed, and simplicity. And as mentioned before, DE also shares many similarities with our implementation. Therefore, SA-DE was an obvious choice to compare our implementation with.

SA-DE has taken advantage of self-adaptive parameter control for the two parameters required for DE. Therefore, the only parameters which are required for SA-DE are the range that CR and F can take on, including the change rate for both of them. We used the same settings as defined in Brest et al. [2006], where cr =change rate:

$F_{cr} = 0.1$, $F_{min} = 0.1$, $F_{Max} = 1$

$CR_{cr} = 0.1$, $CR_{min} = 0$, $CR_{max} = 1$

Standard Evolutionary Algorithm (SEA)

We also implemented a more classical form of EA for comparison. As inspiration for appropriate mutation settings, we looked to Ursem [2002]. Here they use time- and range-scaled Gaussian mutation as expressed in Equation 4.7, where t is the number of generations passed since the search began. The starting value of the global mutation size parameter is set to 10% of the user-defined range, which is presented in Tables 4.1 and 4.2 for each gene individually. For recombination we used N-point crossover with the same settings as AM. For both parent and survival selection, we used the same settings as AM except for an elitism of 10% as opposed to 100%.

$$\sigma = \frac{1}{\sqrt{t+1}} \tag{4.7}$$

4.3 Results

This section presents the results of various tests and experiments. We start out by displaying the results relating to the general performance of the implementations. Then we show the results of some tests regarding specific aspects, such as convergence speed, diversity and the adaptation of the control parameters.

4.3.1 Test Functions

Table 4.5 shows the average score after 5000 generations. The score is averaged over 100 independent runs for each of the GA implementations. The number written in parentheses is the standard deviation of the average score. The results written in **bold blue** are roughly estimated the best results achieved for that function. By roughly estimated we mean the results with the lowest exponent; for example 0.02 has the exponent -2, while 0.002 has the exponent -3 and is thus a better result. But a score of 0.009 will still be roughly estimated to be as good a result as 0.002.

	SEA	DE	SA-DE	AM	AM-KN	AM-KN*
$f1$	0.90 (0.06)	0.60 (3.24)	1e-14 (2e-15)	9e-3 (0.09)	**7e-15**	**7e-15**
$f2$	12.7 (0.93)	1.80 (2.24)	1e-16 (7e-16)	2.31 (2.44)	**0**	0.62 (1.28)
$f3$	74.1 (7.70)	133 (27.1)	1.29 (3.27)	**0.69 (0.08)**	21.7 (15.2)	3.42 (1.57)
$f4$	0.09 (0.02)	0.04 (0.09)	**2e-6 (1e-5)**	0.06 (0.10)	2e-3 (0.02)	0.03 (0.07)
$f5$	431 (182)	68.8 (79.4)	**2e-4**	230 (129)	**2e-4 (1e-5)**	39.2 (63.7)
$f6$	0.47 (0.03)	1e-3 (3e-3)	**7e-5 (7e-4)**	4e-3 (0.02)	**7e-5 (7e-4)**	2e-3 (4e-3)
$f7$	7.64 (4.67)	133 (21.1)	**0.13 (0.04)**	1.23 (1.92)	**0.20 (0.73)**	**0.52 (1.24)**
$f8$	0.53 (0.94)	1.03 (0.88)	**5e-3 (0.04)**	6.31 (3.53)	0.41 (0.86)	4.33 (3.36)
$f9$	90.5 (7.66)	531 (128)	**5e-16 (5e-15)**	77.3 (12.3)	31 (6.12)	53.8 (9.25)
$f10$	3e-6 (2e-6)	0.19 (0.12)	**0**	3e-3 (3e-3)	1e-5 (9e-5)	7e-5 (3e-4)
$f11$	7.89 (0.72)	2e-17 (2e-17)	2e-46 (3e-46)	9e-61 (1e-60)	2e-39 (2e-39)	**1e-67 (2e-67)**
$f12$	5.70 (0.64)	5e-8 (3e-8)	4e-12 (2e-12)	3e-25 (2e-25)	6e-18 (8e-18)	**7e-30 (3e-30)**
$f13$	0.62 (0.07)	8e-5 (4e-5)	2e-5 (9e-6)	4e-10 (2e-10)	2e-7 (5e-7)	**2e-15 (1e-15)**
$f14$	3.67 (3.30)	1.69 (2.27)	**0.97 (1.88)**	5.06 (4.60)	**0.46 (1.19)**	4.83 (4.29)
$f15$	2.32 (0.71)	1.04 (0.50)	**0.40 (0.35)**	4.84 (0.99)	1.80 (0.65)	4.43 (1.00)
$f16$	75.2 (48.5)	**0**	0.03 (0.36)	55.4 (132)	21.6 (25.2)	36 (82.2)
$f17$	18 (1.57)	11 (0.95)	10.5 (0.98)	3.44 (1.61)	1.68 (1.24)	**0.71 (0.96)**
$f18$	4.31 (1.67)	0.12 (0.28)	**0.03 (0.14)**	4.16 (2.39)	1.05 (0.80)	3.96 (1.98)
$f19$	156 (104)	**9.08 (26.1)**	**8.35 (15.3)**	333 (125)	57.4 (45.6)	292 (113)
$f20$	**0**	0.36 (1.13)	**0**	**0**	**0**	**0**
$f21$	167 (46.7)	192 (44.1)	**1.03 (3.67)**	151 (42.7)	23.4 (15.2)	144 (45.5)

Table 4.5: Results from the mathematical benchmark functions, averaged over 100 independent runs. The numbers in parentheses denote the standard deviation. Results written in **bold blue** are the best results for that particular function.

4.3.2 Convergence Speed and Robustness

Table 4.6 shows the success rate of each of the GA implementations. We choose to exclude SEA from this table because of a very low success rate on most of the mathematical benchmarks. How often the GA achieves a satisfying result gives an indication of how robust the algorithm is. We have defined a satisfying result on the test functions as whenever an individual has a genome vector, G, for which Equation 4.8 holds true. This equation measures the Manhattan distance of the genome, G, to the global optimum, O, when the range of the gene, $range_i$, is taken into consideration; this distance has to be less than 10^{-5}.

$$\frac{1}{|G|}\sum_{i\in G}\frac{|g_i - O_i|}{range_i} < \frac{1}{10^5} \tag{4.8}$$

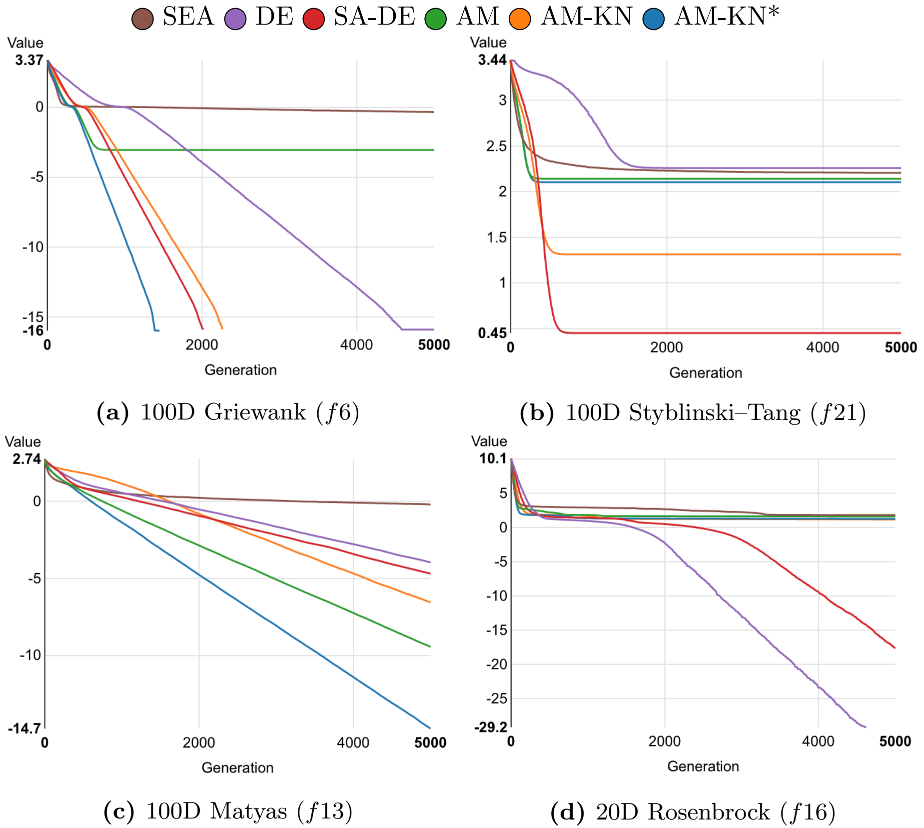
The table also includes the speed of the GA compared to the other implementations. This is noted in parentheses as a multiple of the best GA implementation's

speed. The speed is defined by the average generation where the GA completed the successful runs. Column G denotes the generation which the best GA completed. The algorithm which has the best speed, which is 1, is written in **bold red** .

We can clearly see in Table 4.6 that SA-DE has the best success rate, while AM-KN* has the greatest speed. The trade-off between speed and success rate will be evaluated and discussed in the next chapter. We have also included in Figure 4.4 four fitness plots of some of the benchmark functions to give an overview of the speed and robustness of each the GA implementations throughout the run. All the fitness plots are log_{10} scaled on the y-axis to give a better overview of the fitness plot.

	DE	**SA-DE**	**AM**	**AM-KN**	**AM-KN***	**Best**
$f1$	97% (3.5)	**100%** (1.8)	99% (1.2)	**100%** (1.5)	**100%** **(1)**	**619**
$f2$	50% (3.9)	**100%** (2)	40% (1.2)	**100%** (1.3)	80% **(1)**	**505**
$f3$	0% (0)	0% (0)	0% (0)	0% (0)	0% (0)	**5000**
$f4$	68% (7.2)	**85%** (8.5)	73% **(1)**	19% (7.9)	81% (1.3)	**537**
$f5$	39% (3.5)	**100%** (3)	3% **(1)**	**100%** (2.3)	54% (1.5)	**449**
$f6$	90% (3.3)	**99%** (1.6)	85% (1.2)	**99%** (1.6)	83% **(1)**	**640**
$f7$	0% (0)	0% (0)	66% **(1)**	**84%** (3.9)	**84%** (2)	**948**
$f8$	4% (3.3)	**98%** (1.5)	0% (0)	65% (1.7)	3% **(1)**	**697**
$f9$	0% (0)	**100%** **(1)**	0% (0)	0% (0)	0% (0)	**3548**
$f10$	1% (4.9)	**100%** (2.7)	41% (2.4)	98% (1.3)	95% **(1)**	**980**
$f11$	100% (3.2)	100% (1.7)	100% (1.1)	100% (1.6)	100% **(1)**	**630**
$f12$	100% (3.4)	100% (2.5)	100% (1.4)	100% (1.8)	100% **(1)**	**1181**
$f13$	0% (0)	0% (0)	**100%** (1.5)	67% (1.8)	**100%** **(1)**	**2695**
$f14$	59% (3.5)	75% (2.4)	28% (1.4)	**86%** (1.5)	23% **(1)**	**416**
$f15$	5% (1.9)	**27%** **(1)**	0% (0)	0% (0)	0% (0)	**1278**
$f16$	**100%** **(1)**	99% (1.5)	0% (0)	0% (0)	0% (0)	**2187**
$f17$	0% (0)	0% (0)	3% (1.6)	18% **(1)**	**53%** (2.1)	**1891**
$f18$	85% (1.5)	**95%** (1.5)	1% (1.1)	19% **(1)**	0% (0)	**2497**
$f19$	**85%** (1.8)	74% (2.3)	0% (0)	16% **(1)**	0% (0)	**697**
$f20$	74% (4.3)	91% (1.8)	**100%** (1.1)	**100%** (1.3)	**100%** **(1)**	**218**
$f21$	0% (0)	**94%** **(1)**	0% (0)	18% (1.1)	0% (0)	**954**

Table 4.6:
Success rate from the mathematical benchmark functions, averaged over 100 independent runs. Values written in **bold blue** are the best success rate for that particular function. The numbers in parentheses define the speed of the implementation in comparison to the others, where the values written in **bold red** are the greatest speed achieved. Column "Best" defines the average generation where the implementation with the greatest speed completed.



(a) 100D Griewank ($f6$) **(b)** 100D Styblinski–Tang ($f21$)

(c) 100D Matyas ($f13$) **(d)** 20D Rosenbrock ($f16$)

Figure 4.4: Fitness graphs for four of the benchmark functions, averaged over 10 independent runs. The y-axis denotes the log_{10} scaled fitness value for the best individual in the population, whilst the x-axis defines the generation.

4.3.3 Different Problem Sizes

To get a good perspective of the robustness of a GA, we tested all the GA implementations on different problems sizes for many of the benchmark functions. In Table 4.7, we have included the results when running the benchmark functions on 30 dimensions, and in Table 4.9, the results from running some of the benchmarks on 200 dimensions are shown. We excluded the functions where we already had to adjust the dimensionality to get any of the GA implementations find the global optimum. In Tables 4.8 and 4.10, the related success rate and speed table for 30 and 200 dimensions are included. All the tables are averaged

over 100 independent runs.

	SEA	DE	SA-DE	AM	AM-KN	AM-KN*
$f1$	0.10 (9e-3)	**3e-15**	**3e-15**	**3e-15**	**3e-15**	**3e-15**
$f2$	0.39 (0.04)	**0**	**0**	0.13 (0.56)	**0**	8e-15 (3e-14)
$f4$	4e-4 (8e-5)	**0**	**0**	**0**	**0**	**0**
$f6$	0.02 (6e-3)	**0**	**0**	2e-3 (5e-3)	2e-3 (3e-3)	4e-3 (6e-3)
$f7$	0.04 (7e-3)	28.9 (8.42)	**1e-3 (1e-14)**	**1e-3 (3e-10)**	**1e-3 (4e-6)**	**1e-3 (1e-6)**
$f8$	2e-3 (3e-4)	**1e-32 (5e-48)**	**1e-32 (5e-48)**	**1e-32 (5e-48)**	**1e-32 (5e-48)**	**1e-32 (5e-48)**
$f9$	0.38 (0.05)	78.4 (22.9)	**0**	10.4 (3.17)	1.34 (0.79)	4.23 (2.37)
$f11$	0.04 (8e-3)	2e-55 (2e-55)	1e-104 (3e-104)	**6e-195**	5e-148 (1e-147)	2e-51 (8e-51)
$f12$	0.03 (8e-3)	4e-32 (2e-31)	6e-31 (2e-30)	**0**	**0**	**0**
$f13$	3e-3 (7e-4)	5e-21 (5e-21)	2e-22 (4e-22)	5e-30 (3e-30)	5e-26 (2e-25)	**4e-49 (5e-49)**
$f15$	0.03 (0.09)	4e-53 (7e-53)	**6e-95 (2e-94)**	0.87 (0.35)	0.07 (0.13)	0.69 (0.33)
$f20$	**0**	**0**	**0**	**0**	**0**	**0**
$f21$	11.3 (13.9)	2.83 (8.48)	**0**	8.48 (9.38)	**0**	7.07 (11.4)

Table 4.7: Results from running 30 dimensions on some of the mathematical benchmark functions, averaged over 100 independent runs. The numbers in parentheses denote the standard deviation. Results written in **bold blue** are the best results for that particular function.

	SEA	DE	SA-DE	AM	AM-KN	AM-KN*	Best
$f1$	0% (0)	100% (2.9)	100% (2.3)	100% (1.3)	100% (1.1)	100% **(1)**	**312**
$f2$	0% (0)	100% (3.1)	100% (1.7)	95% (1.7)	100% **(1)**	100% (1.1)	**299**
$f4$	0% (0)	100% (20.9)	100% (8.1)	100% **(1)**	100% (7.7)	100% (1.4)	**181**
$f6$	0% (0)	**100%** (2.9)	**100%** (2.2)	80% (1.3)	75% (1.3)	65% **(1)**	**309**
$f7$	0% (0)	0% (0)	100% (4.8)	100% **(1)**	100% (3.4)	100% (1.2)	**545**
$f8$	0% (0)	100% (3.6)	100% (2.3)	100% (1.5)	100% (1.4)	100% **(1)**	**290**
$f9$	0% (0)	0% (0)	**100%** (2.3)	0% (0)	15% **(1)**	0% (0)	**597**
$f11$	0% (0)	100% (2.9)	100% (2.5)	100% (1.6)	100% (1.2)	100% **(1)**	**281**
$f12$	0% (0)	100% (3.1)	100% (2.7)	100% (1.5)	100% (1.6)	100% **(1)**	**422**
$f13$	0% (0)	100% (2.2)	100% (2.2)	100% (1.6)	100% (1.8)	100% **(1)**	**958**
$f15$	0% (0)	**100%** (3.3)	**100%** (2.7)	0% (0)	75% (1.6)	5% **(1)**	**301**
$f20$	100% (1.6)	100% (4.9)	100% (2.5)	100% (1.1)	100% (1.3)	100% **(1)**	**69**
$f21$	0% (0)	90% (5.9)	**100%** (2.1)	50% (1)	**100%** (1.4)	70% **(1)**	**199**

Table 4.8: Success rate from running 30 dimensions on some of the mathematical benchmark functions, averaged over 100 independent runs. Values written in **bold blue** are the best success rate for that particular function. The numbers in parentheses define the speed of the implementation in comparison to the others, where the values written in **bold red** are the greatest speed achieved. Column "Best" defines the average generation where the implementation with the greatest speed completed.

	SEA	DE	SA-DE	AM	AM-KN	AM-KN*
$f1$	2.20 (0.08)	2.26 (4.08)	5e-8 (2e-7)	2e-12 (9e-13)	4e-6 (9e-7)	**4e-13 (2e-13)**
$f2$	63.9 (2.91)	38 (11.8)	0.13 (0.56)	4.24 (3.34)	**9e-11 (3e-11)**	3.92 (2.98)
$f4$	1.00 (0.18)	0.96 (0.42)	**0.01 (4e-3)**	0.37 (0.24)	**0.06 (0.04)**	0.22 (0.20)
$f6$	1.10 (7e-3)	6e-3 (0.01)	0.01 (0.02)	3e-3 (6e-3)	**1e-9 (3e-10)**	2e-3 (4e-3)
$f7$	91.6 (18.4)	290 (27.8)	20.8 (2.84)	12.7 (5.19)	**3.97 (3.22)**	**6.48 (4.22)**
$f8$	11.4 (4.89)	12.2 (3.83)	**0.16 (0.22)**	28.5 (7.88)	5.56 (3.99)	29.7 (7.49)
$f9$	440 (20)	812 (441)	**72.7 (14.1)**	201 (24.1)	106 (11.4)	172 (17.9)
$f11$	69.5 (4.81)	2e-5 (1e-5)	**3e-24 (9e-24)**	1e-21 (1e-21)	6e-9 (3e-9)	3e-23 (3e-23)
$f12$	44.7 (3.30)	0.09 (0.03)	9e-5 (5e-5)	2e-9 (4e-10)	8e-3 (4e-3)	**2e-12 (9e-13)**
$f13$	4.37 (0.38)	0.21 (0.04)	0.04 (8e-3)	5e-4 (1e-4)	0.96 (0.67)	**6e-6 (2e-6)**
$f20$	7.25 (1.04)	17.4 (11.6)	0.1 (0.44)	**0**	**0**	**0**

Table 4.9: Results from running 200 dimensions on some of the mathematical benchmark functions, averaged over 100 independent runs. The numbers in parentheses denote the standard deviation. Results written in **bold blue** are the best results for that particular function.

	SEA	DE	SA-DE	AM	AM-KN	AM-KN*	Best
$f1$	0% (0)	5% (3.1)	100% (1.2)	100% (1.1)	100% (1.9)	100% **(1)**	**1486**
$f2$	0% (0)	0% (0)	95% (1.2)	10% (1)	**100%** (1.5)	20% **(1)**	**1096**
$f4$	0% (0)	0% (0)	0% (0)	20% **(1)**	0% (0)	**25%** (1.2)	**1353**
$f6$	0% (0)	80% (2.7)	90% (1)	80% (1.1)	**100%** (2)	80% **(1)**	**1544**
$f7$	0% (0)	0% (0)	0% (0)	0% (0)	0% (0)	**15%** **(1)**	**4271**
$f8$	0% (0)	0% (0)	**55%** **(1)**	0% (0)	0% (0)	0% (0)	**1642**
$f9$	0% (0)	0% (0)	0% (0)	0% (0)	0% (0)	0% (0)	**5000**
$f11$	0% (0)	100% (2.8)	100% **(1)**	100% (1.1)	100% (2.1)	100% (1)	**1474**
$f12$	0% (0)	0% (0)	0% (0)	**100%** (1.2)	0% (0)	**100%** **(1)**	**2815**
$f13$	0% (0)	0% (0)	0% (0)	0% (0)	0% (0)	0% (0)	**5000**
$f20$	0% (0)	5% (4)	95% (1)	**100%** (1.2)	**100%** (1.4)	**100%** **(1)**	**599**

Table 4.10:
Success rate from running 200 dimensions on some of the mathematical benchmark functions, averaged over 100 independent runs. Values written in **bold blue** are the best success rate for that particular function. The numbers in parentheses define the speed of the implementation in comparison to the others, where the values written in **bold red** are the greatest speed achieved. Column "Best" defines the average generation where the implementation with the greatest speed completed.

4.3.4 Rotated Solution Space

As discussed in Section 4.1.1, we wanted to test the robustness of the GA implementations when the problems were nonseparable by rotating the solution space. Table 4.11 shows the results from these runs, while Table 4.12 displays the speed

and success rate. Both tables are averaged over 100 independent runs and utilise the same formatting as defined previously.

For most of the benchmarks, both the speed and the success rate of the different GAs are somewhat similar to what we showed in Table 4.6. But we can see there are some benchmark functions where all the GA implementations failed to find the global optimum, and the results are quite bad, with a very high standard deviation.

	SEA	**DE**	**SA-DE**	**AM**	**AM-KN**	**AM-KN***
$f1$	1.62 (0.09)	1e-9 (1e-9)	1e-14 (2e-15)	0.20 (0.42)	**7e-15**	**7e-15**
$f2$	27.1 (4.06)	12.1 (6.40)	**7.23 (4.83)**	36.3 (9.86)	**2.72 (2.92)**	22.6 (8.36)
$f3$	1207 (348)	**971 (316)**	1034 (311)	1247 (312)	1046 (359)	1079 (287)
$f6$	0.71 (0.04)	6e-4 (2e-3)	7e-5 (7e-4)	7e-4 (3e-3)	**3e-11 (4e-11)**	4e-4 (2e-3)
$f8$	1.38 (0.82)	**0.80 (0.64)**	**0.76 (0.64)**	6.67 (3.33)	**0.26 (0.33)**	3.45 (1.88)
$f9$	**210 (41.6)**	**828 (23.2)**	**391 (39.6)**	**593 (324)**	**887 (24.7)**	**867 (22.8)**
$f11$	8.27 (0.52)	2e-17 (1e-17)	2e-46 (9e-47)	9e-61 (3e-60)	2e-39 (3e-39)	**2e-67 (4e-67)**
$f12$	6.89 (0.85)	6e-8 (4e-8)	1e-11 (1e-11)	7e-22 (1e-21)	1e-17 (1e-17)	**3e-28 (6e-29)**
$f13$	1.02 (0.10)	1e-4 (4e-5)	3e-5 (2e-5)	1e-8 (6e-9)	2e-7 (1e-7)	**9e-13 (4e-13)**
$f16$	1138 (2610)	**3.93 (16.4)**	**5.46 (17.3)**	984 (2698)	142 (707)	248 (1251)
$f17$	**23.4 (0.57)**	**23.8 (0.20)**	**23.4 (0.35)**	**23.7 (0.24)**	**23.8 (0.22)**	**23.8 (0.23)**
$f18$	9.10 (3.85)	**0.79 (1.11)**	3.56 (1.95)	5.82 (2.17)	1.92 (1.48)	5.15 (2.07)

Table 4.11: Result from the mathematical benchmark functions with rotated solution space, averaged over 100 independent runs. The numbers in parentheses denote the standard deviation. Results written in **bold blue** are the best results for that particular function.

	SEA	DE	SA-DE	AM	AM-KN	AM-KN*	Best
$f1$	0% (0)	100% (3)	100% (1.5)	81% (1.3)	100% (1.6)	100% **(1)**	**680**
$f2$	0% (0)	1% (1.9)	2% **(1)**	0% (0)	**36%** (1)	0% (0)	**1260**
$f3$	0% (0)	0% (0)	0% (0)	0% (0)	0% (0)	0% (0)	**5000**
$f6$	0% (0)	93% (2.8)	99% (1.8)	92% (1.1)	**100%** (2.3)	95% **(1)**	**947**
$f8$	0% (0)	3% (2.1)	10% (1.2)	0% (0)	**35%** **(1)**	0% (0)	**1490**
$f9$	0% (0)	0% (0)	0% (0)	0% (0)	0% (0)	0% (0)	**5000**
$f11$	0% (0)	100% (3.2)	100% (1.4)	100% (1.1)	100% (1.6)	100% **(1)**	**611**
$f12$	0% (0)	100% (3.2)	100% (2.3)	100% (1.4)	100% (1.7)	100% **(1)**	**1266**
$f13$	0% (0)	0% (0)	0% (0)	**100%** (1.5)	14% (1.6)	**100%** **(1)**	**3108**
$f16$	0% (0)	**68%** **(1)**	23% (1.1)	0% (0)	0% (0)	0% (0)	**4436**
$f17$	0% (0)	0% (0)	0% (0)	0% (0)	0% (0)	0% (0)	**5000**
$f18$	0% (0)	0% (0)	0% (0)	0% (0)	0% (0)	0% (0)	**5000**

Table 4.12: Success rate from the mathematical benchmark functions with rotated solution space, averaged over 100 independent runs. Values written in **bold blue** are the best success rate for that particular function. The numbers in parentheses define the speed of the implementation in comparison to the others, where the values written in **bold red** are the greatest speed achieved. Column "Best" defines the average generation where the implementation with the greatest speed completed.

4.3.5 Pole-Balancing Problem

In Table 4.13, the results of the pole-balancing problem for each of the GA implementations are shown. The table uses the same formatting as we defined for Table 4.5. The score given corresponds to the number of fails during the 10-minute simulation, or an approximation of the number of fails if it reached 200 fails before 10 minutes had passed. A more thorough description of the testing was given in Section 4.1.2.

Table 4.14 shows the success rate and speed of each of the GA implementations. The table uses the same formatting as defined for Table 4.6, apart from the success rate, which is not a rough estimate. To have a successful run, the individual had to be able to balance the pole throughout the whole test.

We ran four different types of the pole-balancing problem, each having a different difficulty level. All the parameters for the test, such as cart weight, pole length and max force, were defined in Section 4.1.2. The four different tests are described as follows:

$p_{1.1}$ = This is the simplest test. Here it is required to balance only a single pole, and the input neurons are mapped directly to the output neurons.

$p_{1.2}$ = Same test as $p_{1.1}$, except that we have included one hidden layer with four neurons. This makes the test harder because the genome of the individual

becomes longer, making it difficult to evolve the parameters because of the ANN's nonseparable nature.

$p_{2.1}$ = This test requires the individuals to be able to balance two poles with different lengths, as defined in Section 4.1.2. The input neurons are mapped directly to the output neurons. This test is harder than $p_{1.1}$ because it both has more input neurons and requires more precision.

$p_{2.2}$ = This test also includes two poles. It is similar to $p_{1.2}$ in that we have included one hidden layer, but the hidden layer in this case has six neurons. Having both two poles and a hidden layer makes this test the most difficult.

	SEA	**DE**	**SA-DE**	**AM**	**AM-KN**	**AM-KN***
$p_{1.1}$	**0**	**0**	**0**	1.84 (12.6)	**0**	**0**
$p_{1.2}$	10.9 (38.2)	**0**	3.92 (23.1)	22.1 (52.1)	**0**	2.08 (14.6)
$p_{2.1}$	10.1 (47.8)	**0**	**0**	0.06 (0.42)	**0**	0.54 (2.56)
$p_{2.2}$	**92.3 (176)**	484 (86.6)	565 (42.7)	146 (207)	280 (205)	**23 (46.3)**

Table 4.13:
Results from running single and double pole-balancing problem, with different numbers of hidden layers. Scores are averaged over 50 independent runs. The numbers in parentheses denote the standard deviation. Results written in **bold blue** are the best results for that particular setup.

	SEA	**DE**	**SA-DE**	**AM**	**AM-KN**	**AM-KN***	**Best**
$p_{1.1}$	100% (1.2)	100% (2.6)	100% (3.7)	96% (1.3)	100% (1.5)	100% **(1)**	**12**
$p_{1.2}$	92% (1)	**100%** (4.3)	92% (5)	84% (1.5)	**100%** (2.6)	98% **(1)**	**58**
$p_{2.1}$	90% (1.3)	**100%** (1.9)	**100%** (3.8)	98% (1.1)	**100%** (1.2)	92% **(1)**	**70**
$p_{2.2}$	6% (1.2)	0% (0)	0% (0)	9% **(1)**	0% (0)	**12%** (1.1)	**386**

Table 4.14: Success rate for 50 runs of the different pole-balancing tests. Values written in **bold blue** are the best success rate for that particular function. The numbers in parentheses define the speed of the implementation in comparison to the others, where the values written in **bold red** are the greatest speed achieved. Column "Best" defines the average generation where the implementation with the greatest speed completed.

4.3.6 Robot Arm Function

Table 4.15 displays the results of the 4 (r_4), 8 (r_8) and 16 (r_{16}) segmented robot arm. The scores are averaged over 100 independent runs. The formatting of the table is the same as in Table 4.5, with the roughly estimated result written in **bold blue** .

	SEA	DE	SA-DE	AM	AM-KN	AM-KN*
r_4	1e-5 (2e-5)	**2e-7 (4e-7)**	1e-4 (1e-4)	5e-6 (5e-5)	3e-6 (7e-6)	3e-6 (9e-6)
r_8	**8e-6 (1e-5)**	2e-4 (6e-4)	2e-4 (1e-4)	9e-3 (0.09)	2e-4 (4e-4)	1e-4 (2e-4)
r_{16}	3e-3 (0.03)	4e-3 (3e-3)	2e-4 (2e-4)	**4e-5 (3e-4)**	2e-3 (2e-3)	8e-3 (0.06)

Table 4.15: Results from running 4-, 8- and 16-segmented robot arm, averaged over 100 independent runs. The numbers in parentheses denote the standard deviation. Results written in **bold blue** are the best results for that particular setup.

	SEA	DE	SA-DE	AM	AM-KN	AM-KN*	Best
r_4	0% (0)	0% (0)	20% (4.6)	**98%** **(1)**	10% (2)	21% (3.1)	**445**
r_8	0% (0)	9% (4.5)	0% (0)	**97%** **(1)**	49% (2.3)	36% (2.8)	**1056**
r_{16}	0% (0)	0% (0)	0% (0)	**68%** (1.1)	4% **(1)**	5% (1.1)	**2665**

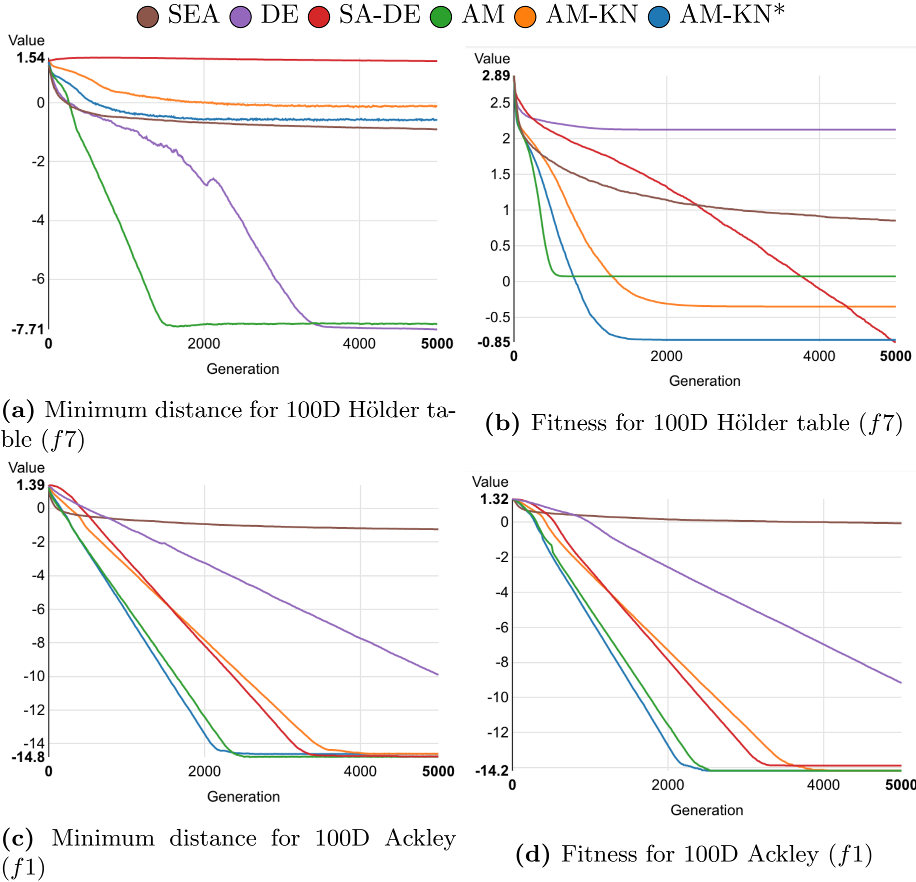
Table 4.16: Success rate and speed from running 4-, 8- and 16-segmented robot arm, averaged over 100 independent runs. The numbers in parentheses define the speed of the implementation in comparison to the others, where the values written in **bold red** are the greatest speed achieved. Column "Best" defines the average generation where the implementation with the greatest speed completed.

Table 4.16 presents the success rate and speed for all the robot arm tests. The speed and success rate are calculated in the same manner as discussed in Section 4.3.2.

We can clearly see AM outperforms the other GA implementations in speed, success rate and results. The reason is to some extent the phenotype and genotype representations of the solutions, which will be discussed in the next chapter, Section 5.1.5.

4.3.7 Diversity

In order to get a picture of the type of exploration that results from different implementations, we have graphed the mean of the distance between individuals and their closest neighbours in the population for two of the benchmark functions. It is worth mentioning that the plot of both mean distance and max distance between individuals are very similar to the minimum distance; therefore, and because we have used the minimum distance in our selection schemes, we chose to plot only the minimum distance. The calculation of the distance between two individuals follows Equation 4.9, where range(g) is the user defined range for gene g.



(a) Minimum distance for 100D Hölder table ($f7$)

(b) Fitness for 100D Hölder table ($f7$)

(c) Minimum distance for 100D Ackley ($f1$)

(d) Fitness for 100D Ackley ($f1$)

Figure 4.5: The graphs on the left, (a) and (c), show the average distance between each individual and its closest neighbour. Their respective fitness graphs, (b) and (d), are on the right. For (a) and (c) the y-axis defines the log_{10} scaled minimum distance between individuals. For (b) and (d) the y-axis defines the log_{10} scaled fitness score. All the graphs are averaged over 10 independent runs.

$$Dist(x, y) = \sum_{g \in G} \frac{|X_{x,g} - X_{y,g}|}{range(g)} \tag{4.9}$$

Figure 4.5 shows the fitness plot both of Ackley ($f1$) and Hölder table ($f7$) (Figures 4.5a and 4.5c) in comparison to the measured minimum distance from

every selected individual to its closest neighbour (Figures 4.5b and 4.5d). Both of the plots are averaged over 10 independent runs of the Hölder table ($f7$) benchmark function and log_{10} scaled along the y-axis.

One can clearly see that the distance plot of Hölder table ($f7$) is distinctly different from its fitness plot, while Ackley ($f1$) has strong similarities between the plots. This is likely caused by the landscape of the benchmark function. $f1$ has one global optimum, while $f7$ has four global optimums, one in each corner of the search space. Having global optimums in each corner gives a high minimum distance if the same genes for two separate individuals are placed in different optimums, even though the fitness is decreasing. For $f1$, on the other hand, all the genes comes closer and closer to the same optimum, causing the minimum gene distance to decrease at the same rate as the fitness score.

Unique Gene Values

Table 4.17 displays statistics about the number of unique gene values for each gene that coexists within the population. The number of unique gene values in a population, P, for any given gene, g, is calculated following Equation set 4.10. Although these equations suggest that the statistics are based upon the actual number of unique genes, it would be just as correct to view these values as the percentage of the maximum possible number of unique genes per gene. This is because the number of individuals in the population, P, happens to be 100 for all the measurements.

$$\begin{aligned} Unique(P,g) &= \sum_{i=1}^{|P|} UniqueVal(i,g) \\ UniqueVal(i,g) &= \left\{ \begin{array}{ll} 0 & \text{if } \forall(j<i)\ X_{i,g} = X_{j,g} \\ 1 & \text{otherwise} \end{array} \right\} \end{aligned} \tag{4.10}$$

$X_{i/j,g}$: The i'th/j'th individuals' value for gene g.

P: The population of individuals.

	AM				AM-KN			
	$Mean$	SD	Min	Max	$Mean$	SD	Min	Max
$Mean_{gen}$	**17.23**	3.59	**8.73**	**26.80**	**27.09**	4.23	**16.85**	**38.09**
Min_{gen}	14.48	1.90	**2.00**	21.00	23.64	1.76	**10.00**	31.00

(a) 10 runs of 100D Sphere ($f11$) (1000 Generations). The average fitness result was 1.2e-08 (Min: 6.2e-09) for AM, and 2.9e-4 (Min: 1.6e-4) for AM-KN

	SEA				DE		SA-DE	
	$Mean$	SD	Min	Max	$Mean$	Min	$Mean$	Min
$Mean_{gen}$	20.84	3.92	11.59	31.21	99.80	**98.87**	99.92	**99.49**
Min_{gen}	17.34	2.90	4.00	25.00	94.07	75.00	97.28	85.00

(b) 10 runs of 100D Sphere ($f11$) (1000 Generations). The average fitness result was 101 (Min: 84.7) for SEA, 6.5 (Min: 4.5) for DE, and (2.0e-05, 7.6e-06) for SA-DE.

	AM				AM-KN			
	$Mean$	SD	Min	Max	$Mean$	SD	Min	Max
$Mean_{gen}$	**15.72**	5.10	**2.92**	26.68	**17.70**	6.70	**3.03**	30.97
Min_{gen}	6.71	2.38	**1.00**	14.00	8.64	2.34	**1.00**	18.00

(c) 10 runs of 100D Hölder table ($f7$) (2500 Generations). The average fitness result was 0.59 (Min: 4.9e-3) for AM, and 0.60 (Min: 6.2e-3) for AM-KN

	SEA				DE		SA-DE	
	$Mean$	SD	Min	Max	$Mean$	Min	$Mean$	Min
$Mean_{gen}$	16.06	4.63	5.84	28.04	**40.96**	**1.41**	60.24	**41.72**
Min_{gen}	12.75	2.85	**1.00**	21.00	5.24	**1.00**	28.62	**5.00**

(d) 10 runs of 100D Hölder table ($f7$) (2500 Generations). The average fitness result was 52.0 (Min: 30.7) for SEA, 131.9 (Min: 102.6) for DE, and 9.8 (Min: 6.0) for SA-DE.

Table 4.17: These tables display the statistics on the number of unique gene values per gene within the population. The numbers from which these statistics are generated are calculated following Equation set 4.10. The first row presents the statistics averaged over all generations in ten distinct runs. The second row shows the minimum values experienced for each of these statistics over the course of the runs. For the particularly interesting cells we have used a **bold** font, and the cells that are mentioned in the text are coloured **blue** .

There are several interesting points to note about these tables, and we have used bold font and colour for the values we found particularly interesting. On Sphere ($f11$), for instance, AM-KN has an average number of unique gene values per gene that is $27.09/17.23 = 1.57$ times higher than that of AM. And the average of the minimum number of unique values found for a gene is $16.85/8.73 = 1.93$ times larger for AM-KN than AM, while for the maximum it is only $38.09/26.80 = 1.42$ times higher. DE and SA-DE have practically no nonunique gene values in the population for a typical generation during runs of Sphere ($f11$). The average minimum number of unique gene values for a gene in the genome is 98.87 for DE and 99.49 for SA-DE, out of a total of 100 possible. Tables 4.17a and 4.17b are definitely better representations of the average results showed for all test functions. In fact, most of the other test functions have close to identical statistics to that of Sphere ($f11$), and this can be seen in the extra tables we have included in appendix A.3. Hölder table ($f7$) had the most deviant statistics, and we see that the deviation is significantly larger for DE and SA-DE than for AM-KN. Meanwhile, the average number of unique genes per gene is reduced only by $17.23 - 15.72 = 1.51$ compared to that of Sphere ($f11$). Note that the average minimum number of unique gene values for a gene is as low as 1.41 for DE on Hölder table ($f7$). This is likely to be caused by the fact that if there is only one value represented for a gene, then no mutation will ever occur to this gene because of the population based mutation scheme in DE.

4.3.8 Adaptive Mutation

One of the actions we have taken to analyse the behaviour of the adaptive control parameters has been to compare the contribution of mutation to that of crossover. We mentioned in Section 3.2.2 that crossover will have an effect on the adaptation of the control parameters. Whenever the value of a gene changes from parent to child, the respective control parameter will adapt to the amount of change that occurred.

Table 4.18 shows how the control parameters have adapted overall. It shows by how much, and in which direction, each operator has influenced the adaptation. Because the size of these parameters usually changes radically over the generations, we needed some way to normalise the changes so that all changes occurring in any generation could be represented. We came to the conclusion that the best way to do this was to divide each change by the average size of all the adaptive parameters in the population before adaptation. It is, however, important to note that changes that are big relative to the size of the respective parameter but small relative to the average size of these parameters do not get the representation they deserve in Table 4.18. The columns in Tables 4.18a to 4.18d display

the following values:

- %: The fraction of control parameters that were adapted during reproduction because of the respective operator(s).
- $\% |Avg|$: The fraction of the total magnitude of adaptation caused by the respective operator(s).
- $\% Avg$: The fraction of the total directed change in the strategy parameters caused by the respective operator(s).
- %+: The fraction of adaptations caused by the respective operator(s) that increased the size of the control parameter.
- $|Avg|$: The average magnitude of change done to the strategy parameters during adaptation caused by the respective operator(s).
- Avg: The average value that was added to the strategy parameters during adaptation caused by the respective operator(s).

If mutation and crossover both played a part in changing the value of a gene during recombination, then the change will be included only in the last row. That is to say that "Mutation" means mutation alone, and "Crossover" means crossover alone. We realise that Table 4.18 might still be difficult to understand; for this reason we have included a more extensive explanation of how these values were calculated and why the overrepresentation of adaptation of large strategy parameters was difficult to avoid in appendix appendix A.4.

	%	$\%\|Avg\|$	%Avg	%+	$\|Avg\|$	Avg
Total	44.88	100	100	43.21	41.08	-5.66
Mutation	**1.38**	3.33	4.4	40.89	44.37	-8.09
Crossover	35.79	78.55	70.75	44.53	40.45	-5.02
Combined	7.7	18.13	24.84	37.49	43.39	-8.19

(a) 10 runs of AM on 100D Sphere ($f11$) (1000 generations). Average fitness achieved: 1.9e-08 (Min: 8.4e-09).

	%	$\%\|Avg\|$	%Avg	%+	$\|Avg\|$	Avg
Total	49.48	100	100	45.72	44.33	-0.99
Mutation	**1.52**	3.51	-19.82	44.6	50.61	**6.45**
Crossover	39.73	77.2	186.11	46.39	42.63	**-2.31**
Combined	8.23	19.29	-66.27	42.73	51.4	**3.99**

(b) 10 runs of AM-KN on 100D Sphere ($f11$) (1000 generations). Average fitness achieved: 2.3e-4 (Min: 1.3e-4).

	%	%$\lvert Avg \rvert$	%Avg	%+	$\lvert Avg \rvert$	Avg
Total	42.78	100	100	44.15	24.99	-3.77
Mutation	**0.04**	0.16	0.49	35.48	39.1	-18.06
Crossover	34.15	80.37	70.99	45.64	25.16	-3.35
Combined	8.59	19.47	28.52	38.27	24.24	-5.36

(c) 10 runs of AM on 100D Hölder table ($f7$) (2500 generations). Average fitness achieved: 0.59 (Min: 4.9e-3).

	%	%$\lvert Avg \rvert$	%Avg	%+	$\lvert Avg \rvert$	Avg
Total	43.14	100	100	47.92	18.32	-3.92
Mutation	**0.13**	1.6	**5.83**	27.74	94.27	**-73.83**
Crossover	34.15	65.35	-17.96	49.51	15.12	**0.89**
Combined	8.85	33.05	112.15	42.1	29.51	-21.47

(d) 10 runs of AM-KN on 100D Hölder table ($f7$) (2500 generations). Average fitness achieved: 1.6e-2 (Min: 5.2e-3).

Table 4.18: These tables show the contributions of mutation and recombination regarding the adaptation of the mutation step size for genes. The meanings of the different rows and columns are presented in the text above (Section 4.3.8). Each table contains data from 10 runs of AM or AM-KN on a single problem. Tables 4.18a and 4.18b contain data from runs on Sphere ($f11$), while Tables 4.18c and 4.18d contain data from runs on Hölder table ($f7$).

Notice that the percentage of change that was caused by mutation alone is significantly lower than what is expected with $P_{mut} = 0.1$ and less than 50% overall. This percentage was expected to be at least $0.1 \times 0.5 = 0.05 = 5\%$, given that there are no advantages from combining mutation and crossover. But in all our results, we see that this number is significantly lower than that, and for Hölder table ($f7$) (Tables 4.18c and 4.18d), the average number of genes that are changed only by mutation is as low as 0.04 for AM and 0.13 for AM-KN. However, it is important to note that even though mutation alone represented only 0.13% of all instances of change for AM-KN on Hölder table ($f7$), it was responsible for 5.83% of the total amount of directed change to the parameters. Another interesting point is the tendency we see for the fraction of mutations that bring about an increase in the step-size parameters. We mentioned in Section 3.2.1 that the expected fraction of values drawn from $\mathcal{N}(0, \sqrt{\frac{2}{\pi}})$ that have an absolute size greater than 1 is about 33%. For Hölder table ($f7$), the deviations from this

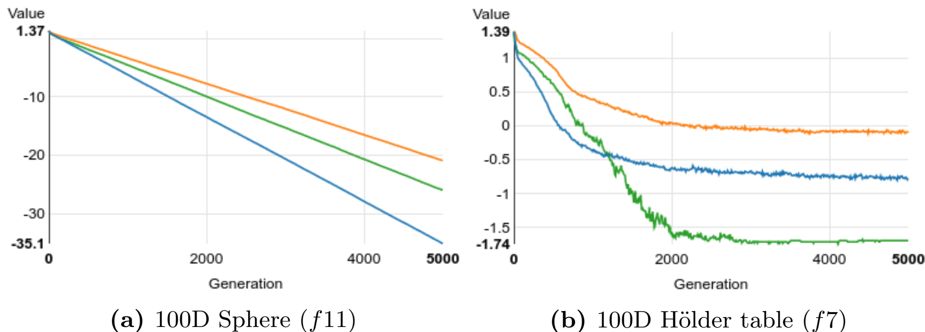
expectation are small, but on Sphere ($f11$), there seems to be a significant deviation with the fractions at 40.89% and 44.6%. We would also like to point out that the direction in which each operator contributes the most is quite different when the selection scheme that promotes diversity is used. On Sphere ($f11$), which is more representative of the more common behaviour of the algorithms, it is clear that for AM all operators contribute in all directions equally. That is to say that the size of their contributions in the positive direction relative to those in the negative direction is about the same. For AM-KN, on the other hand, we see that where mutation is involved, there is an average directed impact on the parameters that is opposite to the directed impact of crossover.

4.4 Contribution to Diversity as an Objective

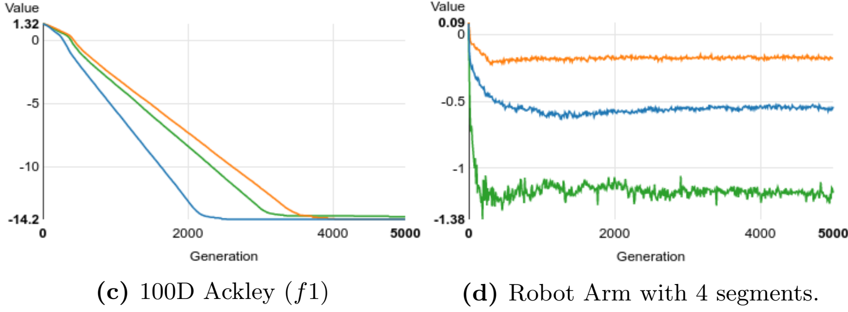
In this section, we present some of the results of running the selection scheme discussed in Section 3.3.4. We will refer to this implementation as AM-CD (adaptive mutation with contribution to diversity). We have chosen to present these results in a separate section because the results showed that this scheme at the very least needs alterations to be able to contribute with something that cannot be gotten from the selection scheme proposed in Section 3.3.3.

AM-CD uses the same setup as AM, but it also requires a set of parameters for the selection scheme. The weight, α, defined in Equation 3.9 was set to 1. The number of survivors was 100, the same as the number of offspring created in a generation.

Table 4.19 displays the success rate and speed of AM-CD in comparison with AM-KN; this table uses the same formatting as defined for Table 4.5. See Section 4.3 for more details on the meaning and calculation of the results in these tables.



(a) 100D Sphere ($f11$) **(b)** 100D Hölder table ($f7$)



(c) 100D Ackley ($f1$) **(d)** Robot Arm with 4 segments.

Figure 4.6: These graphs show the average distance between each individual and its closest neighbour in the population. ● is AM-CD, ● is AM-KN, and ● is AM-KN*. The distance between two individuals is defined by Equation 4.9. The y-axis is the log_{10} scaled average minimum distance between individuals, and the x-axis is the generation number.

	AM	AM-CD	AM-KN	AM-KN*	Best
$f1$	99% (1.2)	**100%** (1.3)	**100%** (1.5)	**100%** **(1)**	**619**
$f2$	40% (1.2)	95% (1.1)	**100%** (1.3)	80% **(1)**	**505**
$f3$	0% (0)	0% (0)	0% (0)	0% (0)	**5000**
$f4$	73% **(1)**	**93%** (2.1)	19% (7.9)	81% (1.3)	**537**
$f5$	3% (1.8)	7% **(1)**	**100%** (4.1)	54% (2.7)	**250**
$f6$	83% (1.2)	90% (1.5)	**99%** (1.6)	83% **(1)**	**640**
$f7$	66% **(1)**	80% (2.3)	**84%** (3.9)	**84%** (2)	**948**
$f8$	0% (0)	33% (1.3)	**65%** (1.7)	3% **(1)**	**697**
$f9$	0% (0)	0% (0)	0% (0)	0% (0)	**5000**
$f10$	41% (2.4)	3% (2.2)	**98%** (1.3)	95% **(1)**	**980**
$f11$	100% (1.1)	100% (1.3)	100% (1.6)	100% **(1)**	**630**
$f12$	100% (1.4)	100% (1.7)	100% (1.8)	100% **(1)**	**1181**
$f13$	**100%** (1.5)	13% (1.8)	67% (1.8)	**100%** **(1)**	**2695**
$f14$	28% (1.4)	63% (3.1)	**86%** (1.5)	23% **(1)**	**416**
$f15$	0% (0)	0% (0)	0% (0)	0% (0)	**5000**
$f16$	0% (0)	0% (0)	0% (0)	0% (0)	**5000**
$f17$	3% (1.6)	25% (2.4)	18% **(1)**	**53%** (2.1)	**1891**
$f18$	1% (1.1)	5% (1.6)	**19%** **(1)**	0% (0)	**2497**
$f19$	0% (0)	0% (0)	**16%** **(1)**	0% (0)	**697**
$f20$	100% (1.1)	100% (1.1)	100% (1.3)	100% **(1)**	**218**
$f21$	0% (0)	8% **(1)**	**18%** (1.2)	0% (0)	**839**
r_4	**98%** **(1)**	43% (1.5)	10% (2)	21% (3.1)	**445**
r_8	**97%** **(1)**	95% (1.1)	49% (2.3)	36% (2.8)	**1056**
r_{16}	68% (1.1)	**85%** **(1)**	4% (1)	5% (1.1)	**2638**

Table 4.19: Success rate of AM-CD in comparison to AM-KN, averaged over 50 independent runs. Values written in **bold blue** show the best success rate for that particular function. The numbers in parentheses define the speed of the implementation in comparison to the others, where the values written in **bold red** are the greatest speed achieved. Column "Best" defines the average generation where the implementation with the greatest speed completed.

Chapter 5

Evaluation and Conclusion

This chapter concludes this report. In Section 5.1, we start by evaluating the data presented in the previous chapter.Section 5.2 concludes the work of this thesis with a retrospective on the research question presented at the beginning. Then at the end, Section 5.3, we will investigate potential future work.

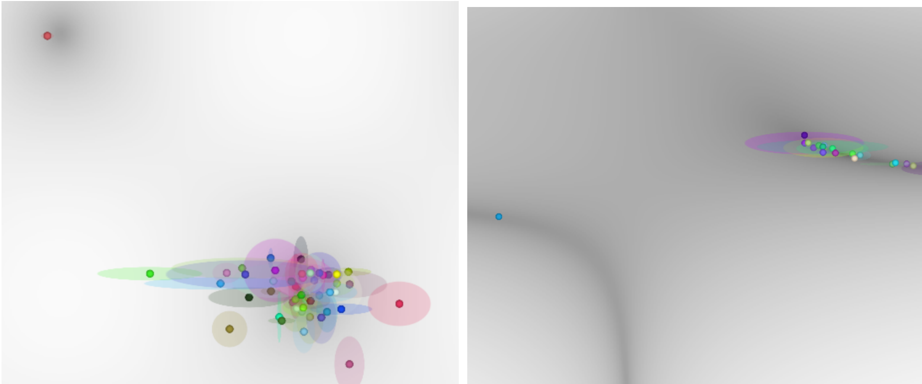
5.1 Evaluation

Looking back at the research questions defined in Section 1.2, the main goal was to utilise information in genomes of individuals to improve guesses for the evolutionary algorithms. This goal has been achieved by using both a gene-specific mutation rate and a diversity-guided selection scheme for the GA, which were discussed in Chapter 3. In this section, we will evaluate the results achieved by this method in comparison to the other implementations presented in the previous chapter. We will endeavour to explain the results from Chapter 4 and investigate important properties of the GA, such as robustness and speed of convergence.

5.1.1 Robustness

As predicted in Section 3.3, our adaptive mutation scheme needs to be complemented with a selection scheme that rewards diversity in order to solve many of the test functions, and it seems that AM-KN has a big contribution in that

respect. However, AM is able to solve more problems than we expected, and when it does find the global optimum, it does so in fewer fitness evaluations than most of the other implementations almost every time. This can be seen in Table 4.6.



(a) AM gene trapped at local optimum on Griewank ($f6$)

(b) AM-KN gene trapped at local optimum on Beale's ($f18$)

Figure 5.1: Illustrates two scenarios where the mutation step size decreases to such an extent that it is impossible for the genes to escape local optimums

Through investigation of what types of benchmarks AM is able to solve, we found that most of the benchmark functions shared the common property of having only one global optimum. This category includes problems such as Sphere ($f11$), Booth's ($f12$) and Matyas ($f13$). We suspect that the reason for this is because of the way our mutation operator converges to optimums, which we will investigate more closely in Section 5.1.2.

AM is often able to find the global optimums of problems such as Griewank ($f6$) and Schaffer N. 2 ($f10$), which have multiple local optimums. This can be seen in Table 4.6. However, to find out what happened when AM converged prematurely, we used the advanced visualisation tool mentioned in Section 3.4.3. Here we saw a tendency that was not so unexpected; one or more of the genes were trapped in a local optimum because of a very small mutation step size for the respective gene(s). This tendency is illustrated in Figure 5.1a, which is taken from the visualisation tool running 100 dimensions of Griewank ($f6$). All the genes except one have flocked around the global optimum, while the last gene is trapped at a local optimum with a close-to-zero mutation size parameter. It is clear that this mutation size is not sufficient to move to a better value for this gene in a single mutation.

In Table 4.5, we see that the robustness of AM-KN is close to that of SA-DE. However, it falls short of SA-DE on somewhat distinct types of functions. While SA-DE finds the global optimum for Rastrigin ($f9$), Levy ($f8$) and Beale's ($f18$), progress stagnates for AM-KN. We can only speculate as to why that is, but an investigation into how AM-KN failed, using the advanced visualisation tool, showed the same tendencies as AM had. One or two genes were simply stuck at the wrong value, with really small values for the mutation operator. Figure 5.1b displays such an example for AM-KN running the Beale's ($f18$) benchmark. One can clearly see that one gene pair is stuck at a local optimum with a very small mutation step size, while the rest of the genes have found the area of the global optimum. The whole ancestral history can be viewed as a video here: http://folk.ntnu.no/thafveli/beale.

All the mathematical test functions presented in Tables 4.1 and 4.2 share a quite specific feature which poses a threat to most adaptive search algorithms. The fitness of a particular genome is the sum of the individual scores for each gene, or pair of genes. Consequently, genes, or gene pairs, are quite often in a state of the search where the size of potential gains in fitness is a lot smaller than the size of potential losses. By being concerned only with cutting losses due to mutation of genes that have already acquired a good value, an adaptive algorithm might prematurely dial down the mutation of the respective gene or genes to such an extent that they are rendered incapable of escaping local minimums once the other genes have caught up with them in the search. This seems to be a bigger problem for AM and AM-KN than for DE and SA-DE. A potential reason for this is that DE never propagates good values for genes without changing them first. Whenever two genomes are merged together in DE, the genes from the first parent's genome remain the same as they were, while all genes from the other parent are mutated. Consequently, it is unlikely that even a single pair of individuals within the population has a single identical-valued gene. This can be seen in Table 4.17b, which displays the statistics on the number of unique gene values during the run of the function Sphere ($f11$). Each individual has to find the same optimum through this combination of mutation and crossover in order for there to be convergence at a single optimum. For AM and AM-KN, on the other hand, if only a single individual fine-tunes a good value inside a local optimum, it is possible for this value to propagate to the entire population. The reason for this is that mutation vectors can be removed from the population if they are unable to produce viable offspring. All mutation vectors might share a single ancestor from only a few generations back. In fact, tests showed that the number of steps back in the chain of ancestors one has to take to find the first common ancestors was often as low as 20, even when the total number of ancestors in the chain was over 1000. That is to say that the first 980 ancestors in the chain of ancestors for each individual in the population were identical; their

evolutionary paths differ only in the last 20 ancestors.

The number of unique genes is a type of diversity that is imperative for the mutation operator in DE. We see that for almost any given generation in the course of a search, there are as many different values for each gene as there are individuals in the population. Since the mutation scheme in AM is not dependent upon having a multitude of different values represented in the population, the search speed can be improved by quickly sharing successful values. However, it might be the case that this type of diversity is the main reason for the increased robustness on the mathematical test functions.

Dimensionality

From Tables 4.7 and 4.9, it is clear that both AM and AM-KN are fairly robust over a wide range of different problem sizes. However, as shown in Table 4.10, the speed of convergence is not as impressive for 200 dimensions as it is for 30 and 100 dimensions when compared to that of SA-DE. We suspect that the constant probability of mutation is the reason for this. For a larger set of dimensions, this can result in too many simultaneous mutations for there to be a realistic likelihood of success. If you throw a die a couple of times in your algorithm, you can call it stochastic, but if you throw it a million times, then you have completely lost the element of surprise in the grand scheme of things. The sum of N independent random variables from a given distribution has a standard deviation that is only $\sqrt{N}$ times bigger than the standard deviation of a single variable. We point this out because it has consequences for our implementation. With a constant mutation rate, the probability of gain in fitness quickly approaches zero for larger genomes. Furthermore, AM's ability to adapt through mutation is decreased significantly. The expected size of the sum of adaptations during a single reproduction is zero. And as the number of dimensions increases, the standard deviation from this mean relative to the size of the mutation vector as a whole approaches zero. This might be the reason why the speed of convergence is less impressive for larger genomes. The first of these problems, the problem of too much simultaneous mutation, would be solved by having an adaptation scheme for the mutation rate as well, like SA-DE. This will be further discussed in Section 5.3.1.

Rotated Solution Space

As discussed in Section 4.1.1, rotating the solution space can create a nonseparable problem out of an originally separable problem. Table 4.11 shows that for

benchmark functions where both AM and AM-KN prematurely converge on the nonrotated problem, it is even worse when the solution space is rotated. Furthermore, the overall speed of convergence is lower. Both DE and SA-DE also perform worse on these problems, but generally not to the same extent, and with a lower standard deviation than that of AM and AM-KN.

In the rotated versions of these problems, a lot more of the genes in the genome need to change simultaneously to make a leap in one of the two-dimensional solution spaces within the rotated solution. Although it is not much to go by, Table 4.12 shows that AM-KN is less reduced in its capability to solve these problems than the others.

5.1.2 Speed of Convergence

The adaptive mutation operator proposed in this thesis has a significantly better speed of convergence than SEA, DE and SA-DE. Table 4.6 in Section 4.3.2 illustrates this point. Notice that AM and AM-KN* are faster than DE and SA-DE on every problem for which they find the global optimum. The number of fitness tests needed to reach the goal defined in Section 4.3.2 is between 1.5 and 6.54 times lower for AM-KN* than for SA-DE whenever they both reach the goal. Between AM-KN* and DE, and AM-KN* and SEA, this ratio is even higher. For 30-dimensional versions of the problems, the gap is even larger. However, as mentioned in Section 5.1.1, the speed of convergence relative to the other implementations is not as impressive on the 200-dimensional problems.

An advantage AM, AM-KN and even SEA have in comparison to DE and SA-DE in speed of convergence is the fact that both DE and SA-DE never propagate good values for genes without changing them first. As mentioned in Section 5.1.1, this trait is likely to increase the robustness for DE and SA-DE on the mathematical benchmarks, but it is also likely to be the reason why AM and AM-KN* need far fewer fitness evaluations to find the global optimum in many cases. Evidence for this can be seen in all the fitness plots in Figure 4.4, where we see both DE and SA-DE start out with less swift progress than the other implementations. At the start of a run on a benchmark, the ability to trade good gene values is at its highest. But it can also be a bad idea to focus on a set of values before the entire solution space has gotten a fair exploration, so it can be both a blessing and a curse.

Another advantage is the ability to increase selection pressure during parent selection. While Self-Adaptive Differential Evolution (SA-DE) and DE, because of their population-based mutation scheme, give all individuals the same number

of reproductions, one can adjust the selection pressure on the more classical implementations of GA and let the better individuals reproduce more frequently.

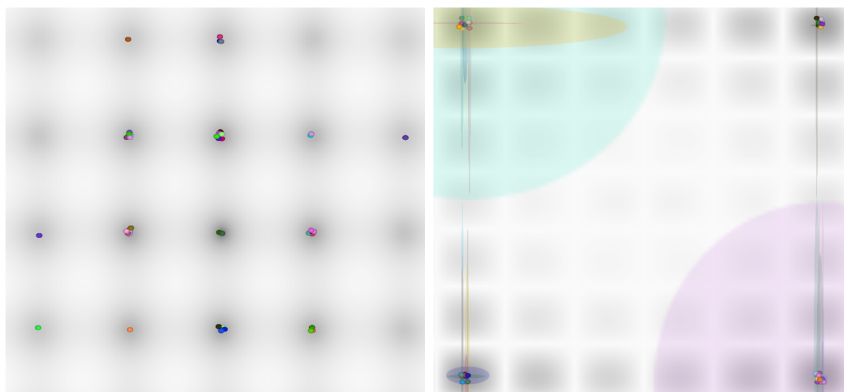
The fast convergence of AM can also be explained by the way the mutation operator is constructed. When only the smaller changes to a gene's value result in successful individuals, the mutation step size of the respective gene decreases in size. After some period of time, the algorithm usually ends up exploring only a single optimum until it has found the bottom, or top, of this particular optimum. As seen in the results, Table 4.5, this is a sufficient technique for problems containing only one optimum, such as Sphere ($f11$), Booth's ($f12$) and Matyas ($f13$). AM is often the fastest to reach the goal, and on average it finds the best solution after 5000 generations.

However, for problems with multiple local optimums, this technique becomes more problematic. We can still see from the fitness plot for Hölder table ($f7$) and Griewank ($f6$), Figures 4.4a and 4.5a, that AM is one of the fastest of the implementations at the beginning of the run, but it sometimes converges on the wrong optimum for one or two of the genes. Looking at the advanced visualisation in Figure 5.2a for the problem Rastrigin ($f9$), we see that AM quickly finds optimums close to the global optimum. And the mutation step sizes decrease very rapidly when these local optimums are found, causing the genes to be trapped in those optimums. The whole ancestral history can be viewed as a video here: http://folk.ntnu.no/thafveli/rastrign. Although we see the same tendency on functions like ackley and griewank, we suspect that it is not a problem to the same extent, because of the general difference in depth of these optimums and how quickly these local optimums are found. For Rastrigin ($f9$) and Levy ($f8$), the differences in fitness score between the top of a peak and the bottom of a valley are huge relative to the difference in fitness score between most local optimums and the global optimum. This is not the case for problems like Ackley ($f1$) and Griewank ($f6$), where there is a much stronger general decline in function value towards the global optimum. Therefore, it is not surprising that finding the bottom of a valley in Rastrigin ($f9$) quickly becomes more attractive, while finding the correct optimum remains attractive for a longer period of time on Ackley ($f1$) and Griewank ($f6$).

Including a selection scheme that promotes diversity to AM, as we have done with AM-KN, slows down the speed of convergence to some degree but helps to avoid premature convergence on many of the problems. The evidence of this can be seen both in Table 4.6 and in the fitness graphs in Figure 4.4. This is not an unexpected behaviour. Converging in general means doing smaller and smaller increments of change. And increased diversity entails a larger overall distance between individuals, which in turn has an effect on the sizes of change that occur during crossover. Furthermore, the likelihood of survival when producing an

individual that is similar to the parent is decreased, which leads to favouring of higher mutation step sizes.

More surprising is the speed of convergence achieved by AM-KN*. When increasing the selection pressure during parent selection, the stalling of convergence in favour of a more diverse population lasts for shorter periods of time. From the results, we see that the speed of convergence is in general even greater for AM-KN* than for AM. We speculate whether this is partially because the selection scheme for survival promotes bigger leaps in solution space. In AM, going for no gain at all through minimal change can be a successful strategy because individuals that reproduce often are the ones with highest fitness to begin with, which might result in passive search strategy. In AM-KN*, on the other hand, being close to another individual is ultimately a bad thing because if an individual is the closest neighbour of another individual, then it has to have a better fitness score than that particular individual in order to survive. However, the impressive speed of convergence might also be caused only by the increased selection pressure.



(a) AM converging at local optimums running Rastrigin ($f9$)

(b) AM-KN slow convergence for Hölder table ($f7$)

Figure 5.2: Visualisation of Convergence

An issue we found with the adaptive mutation scheme is the lack of convergence when the test functions contain multiple global optimums. The genes jump between the different global optimums when a crossover happens between two individuals whose genes are in two different optimums. We could clearly see this effect using the visualisation tool when running both AM and AM-KN on Cross-in-tray ($f4$) and Hölder table ($f7$). Both of these functions have four global optimums. Figure 5.2b shows a run with AM-KN on Hölder table ($f7$) where the global optimums are located in each corner of the solution space. By

looking at the ancestral history of the best individual, we could clearly see that the GA finds the area of the global optimums, which happens quite fast. But the jumping effect mentioned above results in very large mutation step sizes, making it somewhat difficult for the population to converge. The whole ancestral history can be viewed as a video here: http://folk.ntnu.no/thafveli/holdertable. The jumping effect causes the mutation step size to increase, because the mutation step size adapts to the distance the gene has moved from parent to offspring. An increase in the expected size of the mutation makes it difficult for the population to explore a single global optimum to find a better solution. Naturally, this is especially a problem when promoting diversity. Although it is reasonable to keep searching the space for better solutions at this type of range, since good solutions already have proven to be this far apart from one another, it might be better to give up the large change after a while so that convergence may occur. We should say that some exploitation still occurs even for AM-KN and AM-KN*, but from Table 4.6, we can see that the speed of these implementations compared to AM is significantly lower. From Figure 4.5b, one can see that SA-DE maintains a very high diversity throughout its runs on Hölder table ($f7$), and partially because of this, it is much slower to converge and, as a result, never reaches the goal within 5000 generations. It is safe to assume that the high diversity is the result of exploring more than one global optimum for many of the gene pairs.

5.1.3 K-Nearest Neighbours versus Contribution to Diversity

From Table 4.19, we see that the selection scheme presented in 3.3.3, AM-KN, in general performed better than the selection scheme put forth in 3.3.4, AM-CD. Perhaps most surprisingly, AM-CD performed significantly worse on the pole-balancing problem, while it did quite good on the robot arm problem.

We can only speculate why AM-CD is outperformed, but we are starting to believe that despite all the problems with weighing fitness against contribution to diversity, which will be discussed in Section 5.1.3, the real problem is that increased diversity in many cases does not lead to continued progress towards the goal. Some differences among genomes might simply be irreconcilable, and others might be reconcilable but the reconciliation might not contribute to the continued progress of the search. In biological evolution, there is such a thing as a species, and it might be beneficial to use this term in EA as well.

Maintaining diversity contributes to increased robustness, but too much of it can inhibit the algorithm's capability to find any good solution at all. If a solution does not bring about a viable offspring for some period of time, then perhaps

it is best not to pursuit continued exploration of that particular location of the solution space, at least not alongside the other coexisting individuals. This is exactly what happens in AM-KN; diverse solutions are given second chances, but because of the selection pressure during parent selection, each new generation is denser in the areas with greater fitness. Consequently, a location in a solution space that is inferior in terms of fitness is bound to be rejected eventually.

For the problems with multiple optimums, it seems that AM-CD performs quite well compared to AM-KN and AM-KN*. For Cross-in-tray ($f4$), Hölder table ($f7$) and the robot arm problem (r_4, r_8, r_{16}), AM-KN has a hard time converging because of the constant jumping between optimums. AM-CD does not seem to be affected by this problem, or at least not to the same extent. In AM-KN an individual's survival depends on it not being the closest neighbour of a superior individual, and the exact magnitude of the distance does not matter at all. For individuals in AM-CD, it is less important whether it is the closest neighbour and more important how close it is to its closest neighbour within the set of survivors. In other words, the difference between being the closest and the second-closest neighbour of a selected individual can have great significance in AM-KN, even if the second-closest neighbour is almost as close as the closest neighbour. Meanwhile, in AM-CD this would imply only a slight disadvantage, and the disadvantage can be overcome by having a higher fitness than the second-closest neighbour. This type of nuance is the advantage of using the values associated with diversity directly rather than blindly following some rule, as in AM-KN. In Figure 4.6, we can see that for the problems with a single global optimum, AM-CD lies in between AM-KN and AM-KN* in terms of diversity. But in Figure 4.6, we see that the populations in AM-CD are generally less diverse than the populations in both AM-KN and AM-KN*. For the problems where diversity is plentiful, AM-CD has lower diversity than AM-KN and AM-KN*. In Cross-in-tray ($f4$) and Hölder table ($f7$), the distance between two arbitrary individuals is determined largely by the number of genes for which their genes are in different global optimums. The more gene pairs are in different optimums, the longer the distance between the individuals. In AM-KN it is important to always maximise the uniqueness of the selection of optimums in the genome, but for AM-CD fitness becomes increasingly important as the relative increase in contribution to diversity decreases. Whether an individual is at different optimums for 7 or 8 pairs of their genes is not that relevant in AM-CD, but it might be crucial in AM-KN. It is plausible that the performance on the robot arm problem has more or less the same explanation.

Computational Complexity

The results presented in Table 4.19 imply that AM-CD would be the wrong choice for most problems. This is not only due to the results but also because it is more complicated in terms of implementation, and much worse in terms of computational complexity. On the mathematical benchmarks, these selection schemes were the bottlenecks of the implementations. KN does not slow down the run of the algorithm to the same extent as CD. When the solution spaces were rotated, there was not much difference between the running time of AM-KN compared to SEA, DE and SA-DE. AM-CD, on the other hand, was slower by approximately one order of magnitude even for the rotated solution spaces. However, on the pole-balancing problem, the computational complexity of the fitness testing was about two orders of magnitude higher than that of the rest of the algorithm.

Weighing Contribution to Diversity against Fitness

We found that the main difficulty with our approach was to find a balanced weighting between the reward for fitness and the reward for being more distant to the already-selected individuals. We experimented with several ways of weighing diversity and fitness. Our key discovery was that using values for contribution to diversity directly was difficult, if not impossible. Irregularities in the increase in fitness over distances in the solution space contribute to irregularities in the outcome of applying these diversity values directly. Furthermore, most fitness assessments do not present an accurate measure of the utility of different solutions. A fitness score that is ten times better than another rarely means that the solution is ten times as good, if such an idea even makes sense to talk about. For this reason, it is hard to quantify the utility of a small gain in fitness and thus know when and where to value diversity over fitness; when to explore.

As mentioned in Section 4.4, the fitness is set to its inverse, $\frac{1}{Fitness}$, before it is used to calculate the multiobjective score of an individual. The reason for this is that it transforms the fitness landscape drastically. The actual fitness landscapes of minimisation problems are often more similar to Figure 5.3(a) than (b). Using the inverse of the fitness values will shift all fitness landscapes towards being more like Figure 5.3(b) than (a) because a decrease of any size will make a bigger difference to the reverse of the fitness when the initial fitness is low than when the initial fitness is high.

Figure 5.3 illustrates how different fitness landscapes affect the difference between fitness score and contribution to diversity. In Figure 5.3(a), because of the shape

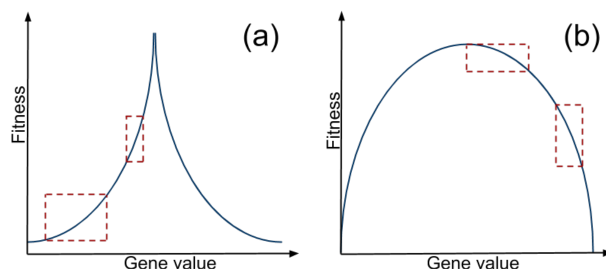


Figure 5.3: Illustrates how different fitness landscapes in local optimums affect the trade-off between fitness and contribution to diversity. In (a) change in gene value relative to change in fitness increases the further away from the optimum an individual is. In (b) the opposite is the case.

of the optimum, spreading out in the basin of the optimum trades a little fitness for a lot of diversity contribution. However, in Figure 5.3(b), the opposite is the case, and the further a solution is from the optimum, the less diversity contribution is gained per unit of fitness lost. In the experiments where we used the actual fitness value, the results were significantly worse. Making an effort to transform the fitness landscape towards that of Figure 5.3(a), as opposed to (b), generally has a positive effect.

We suspect that knowing the global optimum could contribute to setting a fair weight on a difference in fitness based on how far the fitness score is from the score at the global optimum. An additional parameter describing the curvature of the fitness landscape might also add to the ability to set a fair weight. However, in this thesis, we present a generic algorithm which assumes no such information and leave this aspect as possible future work.

5.1.4 Adaptive Mutation

The statistics on the adaptation of the control parameters displayed in Table 4.18 are really interesting. Although mutation occurs to about 10% of genes, as the probability of mutation dictates, most of the mutation occurs along with a crossover. We found this somewhat peculiar and almost as if it had naturally adopted part of the scheme in DE, where crossover always goes hand in hand with mutation and vice versa.

To test whether the separation of mutation and crossover in the scheme was nothing but something to overcome, we tried changing the mutation operator to mutate only genes that also had been subject to a change due to crossover.

The results of the tests were significantly worse than they were without this alteration, and we are not quite sure why this is the case. However, it seems that the mutation size parameters were too high, as opposed to too low, which usually was the case when AM stagnated. This led us to believe that the tendency towards the combination of mutation and crossover is nothing but a strategy that increases the likelihood of survival. It is important to point out that what tends to survive is not synonymous with what contributes to the continued progress of the search. Nevertheless, it might be the case that some tuning with regard to partially joining mutation and crossover would help. More research is needed to pinpoint the effects of coupling mutation and crossover.

Another interesting point in these tables is the percentage of successful mutations that led to an increase in the size of the control parameter. These were significantly higher than expected, and our own prediction had been that the tendency would be towards lower values rather than higher because in the majority of cases, low changes in values are more likely to succeed than higher values. As of now, we have no idea why this is. We suspect that comparing the probability density function for successful values to that of the original distribution, $\mathcal{N}(0, \sqrt{\frac{2}{\pi}})$, might help to figure this out. But due to a shortage of time, this is left as possible future work.

5.1.5 Robot Arm Function

The robot arm problem proved to be quite useful for displaying a particular weakness in both AM-KN and DE. We touched on this particular problem in Section 5.1.2, where we discussed the problem with functions that have multiple global optimums. For the robot arm problem, the difficulty is that not only are there many global optimums, but there is also the problem of measuring the distance between two proposed solutions. If an individual proposes the angle 2π for joint i, while another individual proposes 0, then with respect to this particular genome, they are completely identical in terms of their phenotype. However, a naive distance measure like the one we have used will view these gene values as being as far apart from one another as they can get. It is easy to picture how one can work around this fact, but it does require an implementation that is not oblivious to the specific problem at hand. Furthermore, solutions can be symmetric and thus practically identical even though they appear to be far apart in terms of distance between their genome vectors.

As shown in both Tables 4.15 and 4.16, AM outperforms all the other implementations in both speed and robustness for 4-, 8- and 16-segmented robot arm. The outstanding performance of AM is likely to be caused by the way AM converges

to optimums, as we examined in Section 5.1.2, which seems to be a sufficient technique for the robot arm problem. We suspect that the issue with measuring the distance between solutions is the reason for the poor results on the part of AM-KN compared to its simplified counterpart, AM.

5.1.6 Pole-Balancing Problem

Tests on the pole-balancing problem also yielded some quite interesting results, which are presented in Table 4.13. AM-KN* is definitely the best choice out of these five implementations for these problems. What we find the most interesting, however, is the performance of SEA, DE and SA-DE. SEA is one of the better-performing implementations on this problem; its scores are about as good as that of AM. DE and SA-DE, on the other hand, generally use several times as many generations on average than the other implementations. Considering the significantly better speed of SEA, along with a good success rate, it seems that SEA would be a better option for the pole-balancing problem than both DE and SA-DE, and perhaps also AM-KN. AM-KN trades off speed of convergence for a higher success rate, while AM-KN*, with its increased selection pressure, is the fastest implementation, as well as being robust.

It is worth mentioning that SEA undoubtedly benefits from the fact that finding a satisfying solution to this problem does not require as much precision as it does for the benchmark functions. Consequently, reducing the mutation step size is not necessary for finding a perfect solution. We were suspicious as to whether or not AM-KN* simply performed better because of its increased selection pressure, and so we did additional tests with the same selection pressure for SEA as for AM-KN*. The results from these tests can be seen in Tables A.1 and A.2 in appendix A.2, and they show that SEA does not benefit overall from the increased selection pressure.

5.2 Conclusion

Throughout this thesis, we have investigated methods that can be applied to GAs to improve the guesses for each generation, resulting in fewer fitness evaluations. Looking back at the research questions defined in Section 1.2, we wished to extract and utilise information from the population of genomes to improve the GA. This goal has been achieved through an adaptive mutation scheme that utilises the genetic differences between an offspring and its parent to adapt a mutation vector, along with a selection scheme which promotes diversity. Although the

fundamental principles of DE, as presented by Storn and Price [1997], did not at first seem to be what we were looking for, we found that it exhibits many of the same characteristics. DE utilises information about previously good mutations indirectly, which was discussed in Section 2.1.6. It also preserves the diversity in the population through its parental replacement strategy, combined with a mutative strategy, which by its nature maintains practically 100% diversity in terms of unique genotype values (Section 4.3.7).

The adaptive mutation scheme proposed in this thesis has shown to improve the speed of the GA significantly, which was discussed in Section 5.1.2, and is still able to solve a wide range of test functions as well as the two practical problems. This can be seen in the results presented in Table 4.6. However, as expected, it tends to get trapped in local optimums with mutation control parameters that are too small to make the leap to better areas of the solution space.

Having a selection scheme that maintains diversity is crucial to be able to solve many of the multimodal problems, such as Eggholder ($f5$), Ackley 2D ($f2$) and Levy ($f8$). We have also shown that with a simple diversity-promoting selection scheme, it is possible to increase the selection pressure during parent selection, which can result in an even greater speed of convergence as well as increased robustness, as we reviewed in Section 5.1.2. However, there are many difficulties with measuring the distance between two proposed solutions, and we see one of them in the results for the robot arm problem, as we investigated in Section 5.1.5. Furthermore, we have seen that fitness and diversity within a population are intertwined; converging requires the loss of diversity. For these reasons, and the problem with weighing it against differences in fitness, which is discussed in Section 5.1.3, the selection scheme proposed in Section 3.3.4 (AM-CD) seems to fall short of the simple selection scheme proposed in Section 3.3.3 (AM-KN). However, we still believe that more work on the former can yield fruitful results, which will be discussed in future work Section 5.3.3.

Comparing the results for AM-KN* to SEA, DE and SA-DE and the results in related work, as well as papers that are not mentioned in this thesis, suggests it is a very good implementation of GA with respect to the speed of convergence. Using the adaptive mutation scheme alone often causes the GA to converge prematurely, but supplemented with a simple scheme that maintains diversity, it becomes a good implementation well suited for multiple types of problems, especially those with computationally expensive fitness tests.

5.3 Future Work

Some questions were not answered during our work and are left as future work. Among these are several difficulties regarding measuring and weighing diversity and including additional adaptive control parameters.

5.3.1 Adaptive Rate of Mutation

Parameter tuning can be time consuming, and good adaptation schemes often give better results than even the best-tuned parameter values. Our mutation scheme is adaptive on the part of the size of each mutation, but the rate of mutation is constant. We have tried a few different adaptation schemes for the mutation rate, the simplistic approach in SA-DE included, but in our experience, these control parameters either seem to have little or no effect or tend towards the lowest possible mutation rate. We suspect that a problem with this approach is that mutation rate parameters with a close-to-zero mutation rate simply steal good gene values from the other individuals through crossover, while playing a safe game with respect to exploring completely new values through mutation. If so, it would explain why SA-DE does not seem to have this problem, since it always combines mutation and crossover.

5.3.2 Covariance Matrix

Hansen and Ostermeier [2001] improved the results of their algorithm by adapting a complete covariance matrix for all the parameters evolved. Because of the vast numbers of control parameters to adapt, a large data set seems necessary to get a good approximation of the contours of the fitness landscape. However, a covariance matrix in terms of mutation rate, rather than mutation step size would be somewhat easier to adapt because of the smaller range of values. Specific genes may need to change simultaneously in order to provide a better result, and a low mutation rate inhibits this from happening. A covariance matrix for the rate of mutation could solve this problem.

5.3.3 Weighing Diversity

We have discussed some of the difficulties with measuring and weighing diversity in Section 5.1.3. Some of these may actually have simple solutions whenever the problem and certain features of it are known. For instance, knowing the required

fitness score gives an insight into when to explore and when to exploit. A small difference in fitness is more relevant the closer the individual is to achieving a requirement for satisfaction. Furthermore, we have yet to do tests with different weights for diversity for each individual depending on its fitness score relative to the rest of the population. This could bring about more exploitation in the optimums and more exploration around them, and is inspired by Srinivas and Patnaik [1994].

5.3.4 Phenotype Diversity

Measuring distance purely in the genotype representation has obvious drawbacks, which can be seen in the results for the robot arm problem. However, this does not entail that two solutions cannot be more or less similar. Knowing whether the solution space is circular and symmetric might help for problems such as the robot arm problem.

5.3.5 Steady-State Implementation

There are several perks of steady-state execution, the most important of which is the significantly lower asymptotic limit to speed up with respect to parallel execution. The implementation presented in this thesis ended up with an implementation that almost fits the requirements for steady-state execution. Finding a suitable adjustment which enables steady-state execution could make it more relevant in scenarios where this limit is an issue.

5.3.6 The Island Model

The Island Model is another way to make room for more parallel execution, but it can also be a great way to preserve diversity Sudholt [2015]. Using the adaptive mutation scheme proposed in this thesis along with different Island Model implementations could prove to be more robust and even faster than AM-KN* in the same way AM-KN* outperformed AM. Preserving diversity definitely helps with the exploration of the solution space, but it makes little sense to recombine two individuals whose differences are irreconcilable. In other words, there is a limit to how much diversity a single interbreeding population can take advantage of. Consequently, it makes sense to segregate parts of the population in order to make simultaneous searches at different locations of the solution space.

Bibliography

An, J. and Owen, A. (2001). Quasi-regression. *Journal of Complexity*, 17(4):588 – 607.

Brest, J., Greiner, S., Boskovic, B., Mernik, M., and Zumer, V. (2006). Self-adapting control parameters in differential evolution: A comparative study on numerical benchmark problems. *Evolutionary Computation, IEEE Transactions on*, 10(6):646–657.

Brownlee, J. (2005). The pole balancing problem. *A Benchmark Control Theory Problem. Swinburne University of Technology.*

Cervantes, J. and Stephens, C. R. (2009). Limitations of existing mutation rate heuristics and how a rank ga overcomes them. *Evolutionary Computation, IEEE Transactions on*, 13(2):369–397.

de Jong, E., Watson, R., and Pollack, J. (2001). Reducing bloat and promoting diversity using multi-objective methods.

Deb, K. and Beyer, H.-G. (2001). Self-adaptive genetic algorithms with simulated binary crossover. *Evolutionary Computation*, 9(2):197–221.

Eberhart, R. C. and Kennedy, J. (1995). A new optimizer using particle swarm theory. In *Proceedings of the sixth international symposium on micro machine and human science*, volume 1, pages 39–43. New York, NY.

Eiben, A. E., Hinterding, R., and Michalewicz, Z. (1999). Parameter control in evolutionary algorithms. *Evolutionary Computation, IEEE Transactions on*, 3(2):124–141.

Hansen, N. and Ostermeier, A. (2001). Completely derandomized self-adaptation in evolution strategies. *Evolutionary computation*, 9(2):159–195.

Jamil, M. and Yang, X.-S. (2013). A literature survey of benchmark functions for global optimisation problems. *International Journal of Mathematical Modelling and Numerical Optimisation*, 4(2):150–194.

Janikow, C. Z. and Michalewicz, Z. (1991). An experimental comparison of binary and floating point representations in genetic algorithms. In *ICGA*, pages 31–36.

Jin, Y. (2005). A comprehensive survey of fitness approximation in evolutionary computation. *Soft computing*, 9(1):3–12.

Kao, Y.-T. and Zahara, E. (2008). A hybrid genetic algorithm and particle swarm optimization for multimodal functions. *Applied Soft Computing*, 8(2):849–857.

Kauffman, S. A. (1993). *The origins of order: Self-organization and selection in evolution*, chapter Biological Implications of Rugged Fitness Landscapes.

Kita, H., Ono, I., and Kobayashi, S. (1999). Multi-parental extension of the unimodal normal distribution crossover for real-coded genetic algorithms. In *Evolutionary Computation, 1999. CEC 99. Proceedings of the 1999 Congress on*, volume 2, pages –1588 Vol. 2.

Liu, J. and Lampinen, J. (2005). A fuzzy adaptive differential evolution algorithm. *Soft Computing*, 9(6):448–462.

Lozano, M., Herrera, F., and Cano, J. R. (2008). Replacement strategies to preserve useful diversity in steady-state genetic algorithms. *Information Sciences*, 178(23):4421 – 4433. Including Special Section: Genetic and Evolutionary Computing.

Mallipeddi, R., Suganthan, P. N., Pan, Q.-K., and Tasgetiren, M. F. (2011). Differential evolution algorithm with ensemble of parameters and mutation strategies. *Applied Soft Computing*, 11(2):1679–1696.

Mattiussi, C., Waibel, M., and Floreano, D. (2004). Measures of diversity for populations and distances between individuals with highly reorganizable genomes. *Evolutionary Computation*, 12(4):495–515.

Mengshoel, O. J. and Goldberg, D. E. (1999). Probabilistic crowding: Deterministic crowding with probabilistic replacement. In *Proc. of the Genetic and Evolutionary Computation Conference (GECCO-99)*, pages 409–416.

Nix, A. and Vose, M. (1992). Modeling genetic algorithms with markov chains. *Annals of Mathematics and Artificial Intelligence*, 5(1):79–88.

Ostermeier, A., Gawelczyk, A., and Hansen, N. (1994). A derandomized approach to self-adaptation of evolution strategies. *Evolutionary Computation*, 2(4):369–380.

Pike, R. (2009). The go programming language. *Talk given at Google's Tech Talks.*

Qin, A. K., Huang, V. L., and Suganthan, P. N. (2009). Differential evolution algorithm with strategy adaptation for global numerical optimization. *Evolutionary Computation, IEEE Transactions on*, 13(2):398–417.

Rothlauf, F. (2006). Representations for genetic and evolutionary algorithms. In *Representations for Genetic and Evolutionary Algorithms*, pages 9–32. Springer Berlin Heidelberg.

Shi, Y. and Eberhart, R. (1998). A modified particle swarm optimizer. In *Evolutionary Computation Proceedings, 1998. IEEE World Congress on Computational Intelligence., The 1998 IEEE International Conference on*, pages 69–73.

Srinivas, M. and Patnaik, L. M. (1994). Adaptive probabilities of crossover and mutation in genetic algorithms. *Systems, Man and Cybernetics, IEEE Transactions on*, 24(4):656–667.

Storn, R. and Price, K. (1997). Differential evolution – a simple and efficient heuristic for global optimization over continuous spaces. *Journal of Global Optimization*, 11(4):341–359.

Sudholt, D. (2015). Parallel evolutionary algorithms. In *Springer Handbook of Computational Intelligence*, pages 929–959. Springer.

Ursem, R. (2002). Diversity-guided evolutionary algorithms. In Guervós, J., Adamidis, P., Beyer, H.-G., Schwefel, H.-P., and Fernández-Villacañas, J.-L., editors, *Parallel Problem Solving from Nature — PPSN VII*, volume 2439 of *Lecture Notes in Computer Science*, pages 462–471. Springer Berlin Heidelberg.

Vafaee, F. and Nelson, P. C. (2010). An explorative and exploitative mutation scheme. In *Evolutionary Computation (CEC), 2010 IEEE Congress on*, pages 1–8. IEEE.

Vafaee, F., Turan, G., Nelson, P. C., and Berger-Wolf, T. Y. (2014). Among-site rate variation: Adaptation of genetic algorithm mutation rates at each single site. In *Proceedings of the 2014 Conference on Genetic and Evolutionary Computation*, GECCO '14, pages 863–870, New York, NY, USA. ACM.

Črepinšek, M., Liu, S.-H., and Mernik, M. (2013). Exploration and exploitation in evolutionary algorithms: A survey. *ACM Comput. Surv.*, 45(3):35:1–35:33.

Yang, Z. (1996). Among-site rate variation and its impact on phylogenetic analyses. *Trends in Ecology & Evolution*, 11(9):367 – 372.

Yu, X. and Gen, M. (2010). *Introduction to Evolutionary Algorithms*. Decision Engineering. Springer, Dordrecht.

Appendix A

Appendices

A.1 Videos

In this section a list of videos taken from the GUI of different visualisation tools is presented. The complete list of all the videos have been embedded at http://folk.ntnu.no/thafveli, to make it simpler to view the videos.

Genes of Beale's ($f18$) , http://folk.ntnu.no/thafveli/beal
: This video show a AM implementation trying to solve the 100 dimensional Beale's function in the advanced visualisation tool. It shows how the gene loses their ability to escape global optimums because of a decrease in the standard deviation of the mutation.

Genes of Hölder table ($f7$) , http://folk.ntnu.no/thafveli/holdertable
: This video show a AM implementation trying to solve the 100 dimensional Holder Table function in the advanced visualisation tool. It shows how the gene loses their ability to escape global optimums because of a decrease in the standard deviation of the mutation.

Genes of Rastrigin ($f9$) , http://folk.ntnu.no/thafveli/rastrign
: This video show a AM implementation trying to solve the 100 dimensional Rastrigin function in the advanced visualisation tool. It shows how the gene loses their ability to escape global optimums because of a decrease in the standard deviation of the mutation.

Creating and running an implementation , http://folk.ntnu.no/thafveli/createea

This video show how a new GA implementation in the GUI, by enabling different modules. The video also shows how we can queue tasks for this implementation, and how we select what kind of data that should be stored.

Pole Balancing , `http://folk.ntnu.no/thafveli/polebalance`
This video show the visualisation for the pole balancing problem, viewing a run from a run with AM.

Comparing statistical data , `http://folk.ntnu.no/thafveli/compare`
This videos shows how statistical data from the diffrent GA implementations can be compared in the GUI.

A.2 Graphical User Interface

In this section is a more extensive explanation of the GUI, extending what we was described in Section 3.4.2.

List View

Figure A.1: Gui list

The main view of the GUI is the list view, which is shown if Figure A.1, here we can manage all the runs and different settings which was required to get the statistical data presented in Section 4.3. From the list view we can queue task, view visualisation and show graphs from the different runs.

We have separated between to main types of settings, ea settings and task settings. The EA settings contains the setup for the different ea implementations. The different ea implementations is created by enabling different operators and defining the required parameters, this was reviewed in Sections 3.4.1 and 3.4.2. While the task settings holds the settings for the test we conducted, settings such as dimensions, generation size and the number of generations to run.

Graph Tool

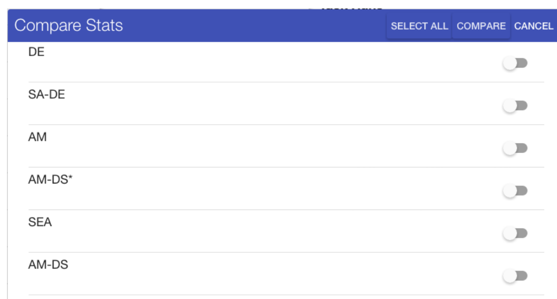


Figure A.2: Comparing statistics between different GA implementations

To enable us to compare the statistical data generated by the different GA implementations we implemented a very simple list whit check-boxes, as shown in Figure A.2, where we can select the algorithm which we want to compare. When the pressing the compare button the graph tool is shown, which can be seen in Figure A.3. Here we have check-boxes for which types of data we want to plot in the graph view. For instance in Figure A.3 we have selected to plot fitness min of all the GA implementations. On http://folk.ntnu.no/thafveli/compare a video showing the process of comparing the statistical data from 50 dimensions of the easom function for all the implementations.

Pole Balancing Problem

Another visualisation tool we implemented is a visualisation for the pole balancing problem. This visualisation renders the run of the best individual when the GA

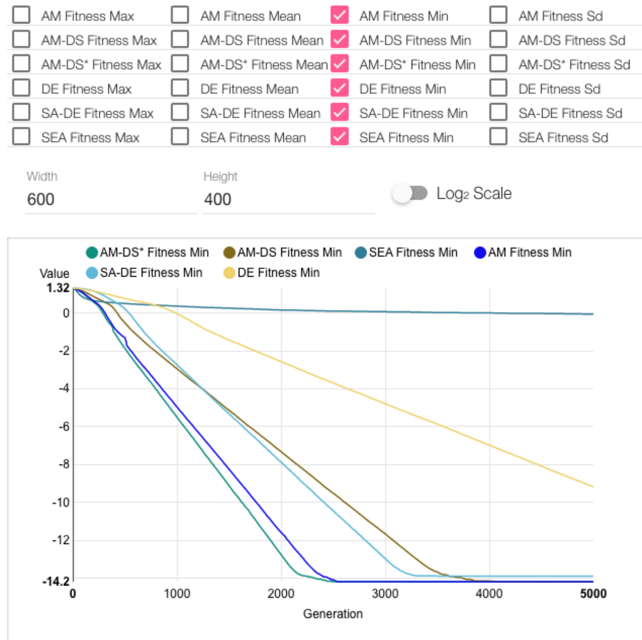


Figure A.3: Graph tool

is done. The visualisation shows the cart and the poles and an arrow which symbolises the force applied to the cart, as can be seen in Figure A.4.

We also included a rendering of the neural network, which can be seen above the cart and poles in Figure Figure A.4. This network shows the the neurons as blue dots and connections between the neurons as lines. The connections is coloured pink if the weight is negative and blue if positive. The width of a connection line is defined by the connection weight. When running the visualisation, which can be done by pressing the play button, the neurons expand and contract depending on output of that neuron. A video from this tool can be viewed at http://folk.ntnu.no/thafveli/poleBalance

This visualisation enabled us to see what happened with both the pole balancing problem and what actually happened in the ANN, which helped us debugging the code, and also made the task more exiting to work with.

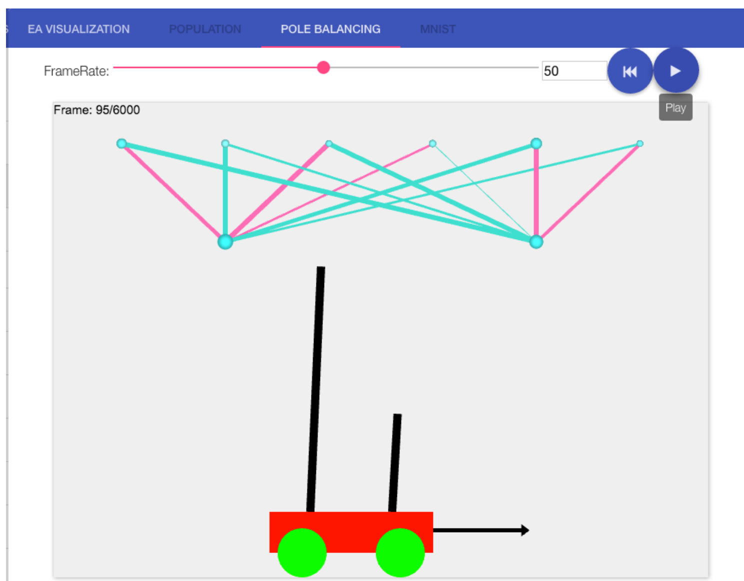


Figure A.4: Visualisation of pole balancing problem

	SEA	SEA*	AM	AM-KN*
$p_{1.1}$	**0**	**0**	1.84 (12.6)	**0**
$p_{1.2}$	10.9 (38.2)	21.2 (52.7)	22.1 (52.1)	**2.08 (14.6)**
$p_{2.1}$	10.1 (47.8)	13.4 (81.7)	**0.06 (0.42)**	0.54 (2.56)
$p_{2.2}$	**92.3 (176)**	238 (290)	146 (207)	**23 (46.3)**

Table A.1: This table shows the result from the tests on Balanving Pole using a higher selection pressure for SEA. SEA* is the version of SEA that has the same selection pressure as AM-KN*

	SEA	SEA*	AM	AM-KN*	Best
$p_{1.1}$	100% (1.4)	100% **(1)**	96% (1.4)	100% (1.1)	**11**
$p_{1.2}$	92% (1.4)	86% **(1)**	84% (1.9)	98% **(1.3)**	**44**
$p_{2.1}$	90% (1.3)	90% (1.9)	98% (1.1)	92% **(1)**	**70**
$p_{2.2}$	6% (1.2)	2% (1.3)	9% **(1)**	**12%** (1.1)	**386**

Table A.2: This table shows the success rates from the tests on an using a higher selection pressure for SEA. SEA* is the version of SEA that has the same selection pressure as AM-KN*.

A.3 Unique Gene Values

In this section we present some of the additional result tables for the statistics on the number of unique gene values. These tables are simply more of the same type of results we presented in Section 4.3.7, though from the runs other test functions.

	AM				AM-KN			
	$Mean$	SD	Min	Max	$Mean$	SD	Min	Max
$Mean_{gen}$	17.08	3.61	8.49	26.68	26.85	4.24	16.58	37.86
Min_{gen}	14.76	2.30	2.00	21.00	22.94	2.11	8.00	31.00

(a) 10 runs of 100D Ackley ($f1$) (1000 Generations). The average fitness result was 1.28e-05 (Min: 7.89e-06) for AM, and 1.5e-3 (Min: 1.3e-3) for AM-KN

	SEA				DE		SA-DE	
	$Mean$	SD	Min	Max	$Mean$	Min	$Mean$	Min
$Mean_{gen}$	20.57	3.92	11.27	30.94	99.22	97.69	97.04	94.41
Min_{gen}	17.67	2.95	4.00	26.00	82.35	63.00	75.46	52.00

(b) 10 runs of 100D Ackley ($f1$) (1000 Generations). The average fitness result was 3.62 (Min: 3.59) for SEA, 0.92 (Min: 0.8) for DE, and (4.5e-3, 1.1e-3) for SA-DE.

	AM				AM-KN			
	$Mean$	SD	Min	Max	$Mean$	SD	Min	Max
$Mean_{gen}$	17.04	3.62	8.37	26.63	26.76	4.28	16.36	37.88
Min_{gen}	13.62	1.79	1.00	21.00	22.52	1.92	8.00	31.00

(c) 10 runs of 100D Levy ($f8$) (1000 Generations). The average fitness result was 7.64 (Min: 5.27) for AM, and 0.15 (Min: 2.93e-05) for AM-KN

	SEA				DE		SA-DE	
	$Mean$	SD	Min	Max	$Mean$	Min	$Mean$	Min
$Mean_{gen}$	21.45	4.05	11.60	32.02	99.86	99.07	99.96	99.35
Min_{gen}	18.39	3.09	4.00	26.00	96.76	80.00	98.97	89.00

(d) 10 runs of 100D Levy ($f8$) (1000 Generations). The average fitness result was 6.96 (Min: 3.76) for SEA, 1.4 (Min: 0.38) for DE, and (1.2e-06, 6.4e-07) for SA-DE.

	AM				AM-KN			
	$Mean$	SD	Min	Max	$Mean$	SD	Min	Max
$Mean_{gen}$	17.93	4.15	8.21	27.17	27.31	4.37	17.58	37.51
Min_{gen}	12.18	1.76	1.00	19.00	21.52	2.33	3.00	29.00

(e) 10 runs of 50D McCormick ($f14$) (1000 Generations). The average fitness result was 5.03 (Min: 1.9e-3) for AM, and 1.26 (Min: 1.9e-3) for AM-KN

	SEA				DE		SA-DE	
	$Mean$	SD	Min	Max	$Mean$	Min	$Mean$	Min
$Mean_{gen}$	19.67	3.88	11.21	28.88	97.65	92.84	97.50	92.20
Min_{gen}	16.56	2.56	3.00	23.00	76.22	42.00	85.84	52.00

(f) 10 runs of 50D McCormick ($f14$) (1000 Generations). The average fitness result was 9.3 (Min: 3.7) for SEA, 1.9e-3 (Min: 1.9e-3) for DE, and 1.26, (Min: 1.9e-3) for SA-DE.

	AM				AM-KN			
	$Mean$	SD	Min	Max	$Mean$	SD	Min	Max
$Mean_{gen}$	18.19	3.82	9.07	28.12	23.90	4.55	12.54	35.44
Min_{gen}	11.31	1.96	1.00	19.00	17.30	2.19	3.00	26.00

(g) 10 runs of 100D Rastrigin ($f9$) (2500 Generations). The average fitness result was 76.01 (Min: 62.68) for AM, and 26.47 (Min: 20.89) for AM-KN

	SEA				DE		SA-DE	
	$Mean$	SD	Min	Max	$Mean$	Min	$Mean$	Min
$Mean_{gen}$	18.98	3.85	9.96	29.22	98.95	95.80	99.83	99.09
Min_{gen}	16.17	2.58	3.00	23.00	90.04	66.00	93.94	77.00

(h) 10 runs of 100D Rastrigin ($f9$) (2500 Generations). The average fitness result was 265 (Min: 250) for SEA, 790 (Min: 776) for DE, and 37.9 (Min: 29.9) for SA-DE.

	AM				AM-KN			
	$Mean$	SD	Min	Max	$Mean$	SD	Min	Max
$Mean_{gen}$	17.26	5.33	2.19	27.61	22.69	6.09	6.17	35.15
Min_{gen}	10.46	0.88	1.00	19.00	16.32	1.00	1.00	24.00

(i) 10 runs of 50D Easom ($f17$) (2500 Generations). The average fitness result was 4.6 (Min: 1.0) for AM, and 1.3 (Min: 0.0) for AM-KN

	SEA				DE		SA-DE	
	$Mean$	SD	Min	Max	$Mean$	Min	$Mean$	Min
$Mean_{gen}$	24.85	6.91	8.81	37.21	81.33	62.43	81.83	64.08
Min_{gen}	18.86	2.69	2.00	28.00	70.58	34.00	73.36	40.00

(j) 10 runs of 50D Easom ($f17$) (2500 Generations). The average fitness result was 19.3 (Min: 17.0) for SEA, 17.9 (Min: 16.3) for DE, and 17.8 (Min: 17.0) for SA-DE.

A.4 Adaptive Mutation Results

This section contains a more detailed description of the values presented in Table 4.18 and Section 4.3.8, as well as result tables from run of different problems. Equation set A.1 contains the mathematical definitions of the values presented in Table 4.18.

$$
\begin{aligned}
f(r, op) &= \left\{ \begin{array}{ll} 1 & \text{If operator } op \text{ was the cause} \\ & \text{of change in reproduction } r. \\ 0 & \text{Otherwise} \end{array} \right\} \\
\%_{op} &= \frac{\sum f(r, op)}{|R|} \\
\% \left|Avg\right|_{op} &= \frac{\sum |A_r| \, f(r, op)}{\sum |A_r|} \\
\% Avg_{op} &= \frac{\sum A_r f(r, op)}{\sum A_r} \\
\% +_{op} &= \frac{\sum f(r, op) pos(r)}{\%_{op}} \\
\left|Avg\right|_{op} &= \frac{\sum f(r, op) \, |A_r|}{\sum f(r, op)} \\
Avg_{op} &= \frac{\sum f(r, op) A_r}{\sum f(r, op)}
\end{aligned} \tag{A.1}
$$

$\sum$: $\sum_{r \in R}$, is the total set of all new genes that have been created throughout the run of the algorithm on a problem.

A_r: The adaptation, the change, in reproduction r, relative to the average size of the control parameters in the population before adaptation.

op: Either "Mutation", "Recombination", "Both", or "Total". f(r,"Total") is true whenever an adaptation has taken place, $f(r, Total) = 1 \iff A_r \neq 0$. However, only one of f(r,"Mutation"), f(r,"Crossover"), and f(r,"Combined") is 1 for the same r.

Unfortunately we could not simply use the relative change, meaning the size of the change relative to the size of the parameter, because all changes that decrease the parameters are smaller relative to the initial parameter than changes that increase the parameters. And sadly it does not help to use the change relative to the new parameter value, nor the change relative to the smallest of the old and the new parameter. However, we came to the conclusion that the best way to do this was to divide each change by the average size of all adaptive parameters within the population before adaptation. Although changes to the larger parameters within the population still get overrepresented in this normalisation scheme, it is beneficial because it makes it possible to generalise over thousands of generations. Tables A.4c and A.4d are examples of how this normalisation is imperfect to say the least. We see that the average size of change is one order of magnitude larger than the average size of the mutation parameters in the population. Easom ($f17$) has a flat fitness landscape with tiny drops around (π, π), and whenever a gene pair has found this valley their mutation size parameters must drop drastically in order to pinpoint the exact position of the optimum. Consequently there should be a tremendous variation in the size of the mutation size parameters, and evidently this seems to be the case.

Below are some additional result tables from runs of different test functions.

	%	%$\lvert Avg \rvert$	%Avg	%+	$\lvert Avg \rvert$	Avg
Total	44.91	100	100	43.33	41.36	-5.39
Mutation	1.37	3.28	4.46	40.87	44.53	-7.88
Crossover	35.81	78.59	70.1	44.68	40.77	-4.73
Combined	7.74	18.12	25.44	37.53	43.52	-7.96

(a) 10 runs of AM on 100D Griewank ($f6$) (750 generations). Average fitness achieved: 1.68e-05 (Min: 7.71e-06).

	%	%$\|Avg\|$	%Avg	%+	$\|Avg\|$	Avg
Total	49.49	100	100	45.67	44.38	-0.97
Mutation	1.51	3.47	-19.06	44.46	50.32	6.1
Crossover	39.74	77.27	185.49	46.34	42.7	-2.25
Combined	8.24	19.26	-66.41	42.65	51.37	3.9

(b) 10 runs of AM-KN on 100D Griewank ($f6$) (750 generations). Average fitness achieved: 2.79e-2 (Min: 1.19e-2).

	%	%$\|Avg\|$	%Avg	%+	$\|Avg\|$	Avg
Total	42.04	100	100	46.08	39.79	0.1
Mutation	0.35	2.18	-268.14	41.23	105.08	-31.21
Crossover	33.32	78.61	1826.42	47.78	39.46	2.22
Combined	8.37	19.21	-1458.26	39.52	38.37	-7.04

(c) 10 runs of AM on 50D Easom ($f17$) (1500 generations). Average fitness achieved: 9.47 (Min: 6.0).

	%	%$\|Avg\|$	%Avg	%+	$\|Avg\|$	Avg
Total	44.21	100	100	46.89	37.99	0.09
Mutation	0.44	1.99	-498.46	30.57	75.51	-42.72
Crossover	35.44	80.52	2410.13	48.42	38.16	2.58
Combined	8.33	17.48	-1811.65	41.21	35.27	-8.26

(d) 10 runs of AM-KN on 50D Easom ($f17$) (1500 generations). Average fitness achieved: 3.6 (Min: 2.4e-3).

	%	%$\|Avg\|$	%Avg	%+	$\|Avg\|$	Avg
Total	44.59	100	100	42.3	41.66	-6.23
Mutation	1.47	3.25	3.03	40.7	40.94	-5.72
Crossover	35.41	77.8	80.23	43.3	40.81	-6.3
Combined	7.7	18.95	16.74	37.99	45.72	-6.04

(e) 10 runs of AM on 100D Ackley 2D ($f2$) (1000 generations). Average fitness achieved: 2.14 (Min: 6.71e-07).

	%	%$\|Avg\|$	%Avg	%+	$\|Avg\|$	Avg
Total	48.97	100	100	44.38	45.46	-2.49
Mutation	1.61	3.95	-11.95	44.04	54.73	9.13
Crossover	39.19	76.34	154.99	44.74	43.37	-4.83
Combined	8.18	19.71	-43.02	42.72	53.66	6.44

(f) 10 runs of AM-KN on 100D Ackley 2D ($f2$) (1000 generations). Average fitness achieved: 9.39e-05 (Min: 7.02e-05).

	%	%$\|Avg\|$	%Avg	%+	$\|Avg\|$	Avg
Total	44.02	100	100	40.41	39.64	-9.16
Mutation	0.19	0.49	0.39	42.19	44.2	-8.17
Crossover	35.1	78.86	71.6	41.75	39.2	-8.23
Combined	8.73	20.65	28.01	35	41.29	-12.95

(g) 10 runs of AM on 50D McCormick ($f14$) (500 generations). Average fitness achieved: 4.71 (Min: 1.9e-3).

	%	%$\|Avg\|$	%Avg	%+	$\|Avg\|$	Avg
Total	49.05	100	100	44.08	42.69	-3.15
Mutation	0.36	0.98	-3.79	51.24	56.85	16.33
Crossover	39.36	77.57	103.81	44.79	41.27	-4.08
Combined	9.33	21.45	0	40.79	48.15	0

(h) 10 runs of AM-KN on 50D McCormick ($f14$) (500 generations). Average fitness achieved: 0.32 (Min: 1.9e-3).

	%	%$\|Avg\|$	%Avg	%+	$\|Avg\|$	Avg
Total	43.41	100	100	43	37.34	-6.34
Mutation	0.77	2.07	3.56	39.22	43.44	-12.68
Crossover	34.6	79.29	73.69	44.33	37.14	-5.86
Combined	8.04	18.63	22.75	37.62	37.58	-7.79

(i) 10 runs of AM on 100D Rastrigin ($f9$) (750 generations). Average fitness achieved: 86.86 (Min: 75.62).

	%	$\%\|Avg\|$	%Avg	%+	$\|Avg\|$	Avg
Total	45.91	100	100	45.65	44.51	0.14
Mutation	1.31	3.39	-0.9	42.77	52.75	-0.03
Crossover	36.56	76.24	-831.74	46.28	42.61	-1.43
Combined	8.04	20.38	932.66	43.26	51.81	7.35

(j) 10 runs of AM-KN on 100D Rastrigin ($f9$) (750 generations). Average fitness achieved: 102.47 (Min: 74.1).