AN INVESTIGATION INTO THE EFFECTIVENESS OF STOCAHSTIC OPTIMISATION ALGORITHMS IN PLAYING CONNECT 4

by

CHRISTOPHER PHILLIPS
URN: 6085214

A dissertation submitted in partial fulfilment of the
requirements for the award of

BACHELOR OF SCIENCE IN COMPUTER SCIENCE

May 2013

Department of Computing
University of Surrey
Guildford GU2 7XH

Supervised by: James Heather

I declare that this dissertation is my own work and that the work of others is acknowledged and indicated by explicit references.

Christopher Phillips
May 2013

Abstract

There has been much research into teaching computers to play games effectively, resulting in programs which are capable of beating the best human players at chess and other board games. These programs typically make use of strongly defined rules to provide their intelligence. This project suggests a different technique; applying stochastic optimisation techniques to effectively learn the strategic rules for playing Connect 4.

Three popular algorithms, a genetic algorithm, evolution strategies and particle swarm optimisation, were implemented to play Connect 4 in The Arena, a Java based framework providing the implementation of the game. Following a training period to learn the rules of the game, the playing performance of these algorithms was tested. The test results showed limited success, with only one of the algorithms able to play relatively successfully. The time constraints for the development of this project may be the reason for the poor performance of the algorithms, therefore more research and testing should be considered.

Acknowledgements

With thanks to Dr James Heather, for his help during the development of this project, and Dr Johann A. Briffa, for no doubt saving the author countless arguments with word processing software whilst trying to format this dissertation.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

As computational power has increased, research into techniques which allow a computer to simulate intelligence has boomed. Investigating techniques for teaching computers to play games as good as, or better, than humans is a popular research area. This sort of research is interesting in an academic sense, but could also find applications in modern video games, where there is a demand for ever more realistic artificial intelligence to provide a challenging experience for players. Research in this area has had some successes, with computers now able to beat the best human grandmasters at chess and several other board games.

Connect 4 was solved in 1988 by Victor Allis and he described a program, VICTOR, which plays Connect 4 according to nine strategic rules and is capable of always winning[1]. This is a similar approach taken to other successful game playing programs; their 'intelligence' is based on a set of strongly defined rules for playing the game.

This project aims to take a different approach to playing Connect 4; applying stochastic optimisation computational intelligence techniques to attempt to learn the best way of playing Connect 4, essentially learning the strategic rules.

1.1 Project Aims

The aim of this project is to investigate the application of population based optimisation techniques to playing Connect 4 in The Arena, a Java based framework that allows player modules to be automatically played against each other. Specifically, three algorithms will be investigated:

- Genetic Algorithms

- Evolution Strategies
- Particle Swarm Optimisation

This aim was split into a number of objectives, which will be used in evaluating the success of the project (see chapter 5). These objectives are listed below.

- Implement and test the chosen population based optimisation algorithms, producing a generic implementation
- Adapt the generic implementation to work with Connect 4 in The Arena
- Evaluate the effectiveness of the different algorithms in playing Connect 4
- Compare the effectiveness and efficiency of the different algorithms

1.2 Report Structure

This report will detail the research, design and testing for the project and is structured as follows:

- Chapter 1: introduction. A brief description of the project and a listing of the project's aims and objectives
- Chapter 2: literature review. The research carried out for the project. Describes the structure and operation of the three chosen optimisation algorithms
- Chapter 3: system specification. Lists the requirements of the system to be developed and gives details of the development methodology employed and the time plan for the development process
- Chapter 4: system design. Details and discusses the design of the system and the testing processes. Gives details of any implementation problems and how they affected the design
- Chapter 5: evaluation. Presents the results of testing the system and evaluates the success of the project against the objectives
- Chapter 6: conclusion. A final discussion on the successes and limitations of the project and suggestions for future work

Chapter 2

Literature Review

2.1 Optimisation

In simple terms, mathematical optimisation, also known as mathematical programming, is the process of finding the optimal solution to some function based on a set of constraints. Optimisation problems typically take the form[2]:

$$\begin{aligned} &\text{maximise (minimise) } f(x) : x \text{ in X} \\ &\text{subject to:} \\ &\quad X \in \mathbb{R} \\ &\quad g(x) < 0 \\ &\quad h(x) = 0 \\ &\text{where:} \\ &\quad \text{X is the set of values for } x\text{; the domain of the function} \\ &\quad f(x) \text{ is the function being optimisied; the objective function} \\ &\quad g(x),\ h(x) \text{ are constraints on } x \\ &\quad \text{The number of contraints depends on the optimisation problem} \end{aligned} \tag{2.1}$$

The goal of optimisation techniques is to "find the peak of the fitness landscape"[3]. The fitness landscape, or search space, of a function is a plot of the solutions to the function against their fitness value. Each solution is a feasible solution to the function and the fitness of a solution is a measure of its quality, i.e. the result of the objective function. See fig 2.1[4] for an example fitness landscape for the 2D sphere function.

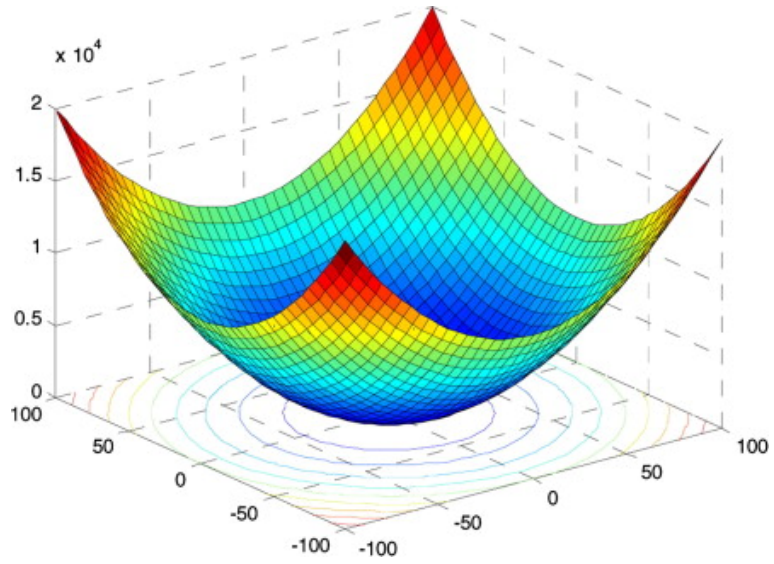


Figure 2.1: Fitness landscape for the 2D sphere function

For this project, one type of optimisation technique was studied; stochastic optimisation.

2.1.1 Stochastic Optimisation

Stochastic optimisation algorithms make use of randomness in order to move through the fitness landscape of the function. This gives this class of algorithm an advantage over more traditional optimisation techniques when dealing with real world problems.

Traditional optimisation techniques, such as gradient descent, are deterministic and guarantee to find the optimum solution when dealing with relatively simple problems. However, they are often computationally infeasible on many real world problems or may be unable to optimise the function at all if many local optima are present in the fitness landscape (see fig 2.2[5] for an example fitness landscape with local optima).

The random aspect of stochastic optimisation means that these algorithms only sample a portion of the solution space which makes them much more computationally feasible on large continuous problems. The randomness also means that stochastic algorithms are capable of "jumping" around the solution space, which allow them to deal with the problem of local optima.

A wide variety of stochastic algorithms are available and due to the author's interests, those chosen for this project are computational intelligence techniques.

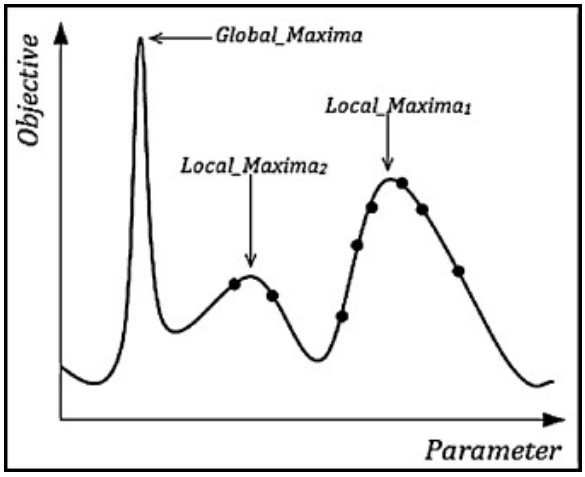


Figure 2.2: A simple fitness landscape showing 2 local optima

2.2 Computational Intelligence

Computational Intelligence (CI) is often used in general to refer to artificial intelligence (AI) techniques, but is actually a distinct branch of AI. AI is defined by John McCarthy, considered to have coined the term in 1955, as "the science and engineering of making intelligent machines"[6]. It is more commonly defined as "the study of the design of intelligent agents"[7].
CI techniques are designed to replicate observable intelligence mechanism in nature, for example simulating the ways in which the human brain works or the collective intelligence of an ant colony.
This project looked at two CI techniques:

- Evolutionary Algorithms
- Swarm Intelligence

Both of these techniques are population based, this means they use a population of solutions which are then manipulated to move through the fitness landscape of the function.

2.3 Evolutionary Algorithms

Evolutionary Algorithms (EAs) are a set of algorithms designed around the principles of natural evolution and Darwinian selection. Carlos A. Coello Coello describes the general structure of an

EA as "a population of encoded solutions (individuals) manipulated by a set of operators and evaluated by some fitness function."[8]

2.3.1 Encoding

Solutions, or individuals, in EAs and other population based algorithms are encoded. This involves mapping the "features" of a solution, i.e. the parameters of the objective function, into a machine readable format. The "features" of a solution are also known as decision variables. This machine readable form is called the *genotype*, and is analogous in nature to DNA. As with DNA, the genotype is expressed (decoded) to give the features, or *phenotype*, of the individual. An individual's genotype is made up of a number of *chromosomes* and each chromosome consists of a number of *genes*. As with nature, the individual genes are responsible for encoding some feature in the phenotype. With EAs, each gene typically encodes one phenotype feature, however it is also possible for one gene to encode multiple features, *pleiotropism*, or for a single feature to be encoded by multiple genes, *polygeny*.
The two most common methods of encoding are:

- Binary coding. Each chromosome is represented by a string of binary bits
- Real value coding. Each chromosome is represented by a string of real values, i.e. the actual phenotype values

Other encoding methods include graph/tree encoding.

2.3.2 Operators

Genetic operators are the methods by which individuals in EAs are moved through the fitness landscape. They are designed to imitate natural processes and introduce *variance* into the population.
Three operators are used in EAs, and each algorithm provides their own implementation:

- Selection/Reinsertion. Used to select individuals from the population. Often imitates Darwinian selection; those individuals that are most fit are more likely to survive
- Recombination. Used to produce new individuals from a number of parent individuals; imitates breeding

- Mutation. Used to provide further variance in the genotype of individuals; imitates genetic mutation

2.3.3 Generic Evolutionary Algorithm

The structure of EAs is typically the same, first an initial (parent) population is generated. This population is evaluated using the fitness function, and if stopping criteria are met the algorithm exits. If the criteria are not met, then a number of offspring individuals are generated, using selection, recombination and mutation operators. These offspring individuals are evaluated with the fitness function and the reinsertion operator combines the parent and offspring individuals into the population. The stopping criteria are checked again, and if they are still not met this process repeats. See fig 2.3 for a diagram[9], and related pseudocode[10], of this process.
As mentioned, the stopping criteria are used to end the EA. The simplest and most common form is a limit on the number of iterations for the algorithm; this is almost always included. Another common stopping criteria is to stop when the result of the fitness function is greater than (or less than, depending on maximisation or minimisation) some defined constant.

2.3.4 Specific Evolutionary Algorithms

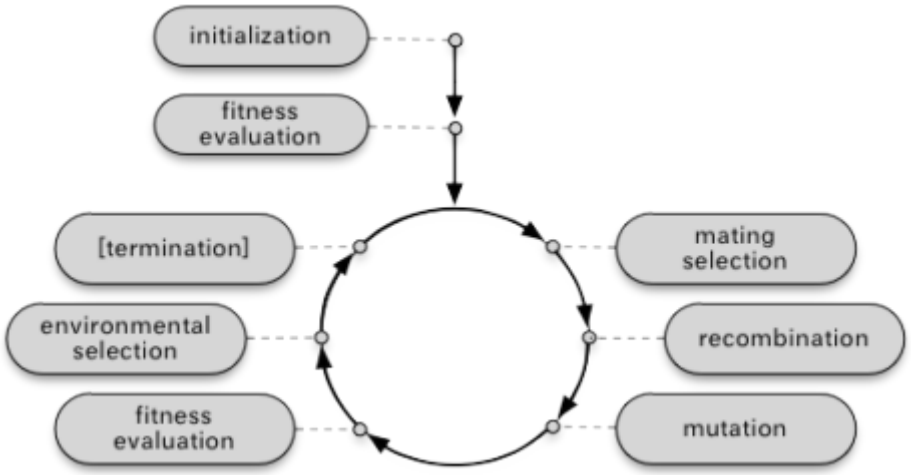
While the section above discussed the generic operation of EAs, this section will detail the specific EA algorithms investigated for this project.
There are three main EA variants:

- Genetic Algorithms
- Evolution Strategies
- Genetic Programming

The genetic and evolution strategies algorithms were studied for this project, and will now be discussed.

2.3.5 Genetic Algorithms

Genetic Algorithms (GAs) are some of the earliest examples of EAs. First suggested in the late 1950s and early 1960s, these initial algorithms attracted little follow up. It was not until 1975



(a) Generic EA diagram

```
t = 0;
initialize(P(t=0));
evaluate(P(t=0));
while isNotTerminated() do
    P_p(t) = P(t).selectParents();
    P_c(t) = reproduction(P_p);
    mutate(P_c(t));
    evaluate(P_c(t));
    P(t+1) = buildNextGenerationFrom(P_c(t), P(t));
    t = t +1;
end
```

(b) Psuedocode for generic EAs

Figure 2.3: Diagram and pseudocode for the operation of a generic EA

that interest in GAs took off, with John Holland's book *Adaptation in Natural and Artificial Systems*[11–13]

GAs are designed to model genetic evolution and use the selection and recombination operators as the main evolutionary force[11].

2.3.5.1 Encoding

The canonical GA, as defined by John Holland, use binary coding to encode the individuals in the population; most GA variants use this encoding method (see section 2.3.1).

The use of binary coding allows the recombination and mutation operators to be very simple

but makes the evaluation of an individual's fitness slightly more complex. Before evaluation can be performed, the individual's genotype must be converted (decoded) into its phenotype. An individual I, with n_x features will be encoded as: I = (b_1, ..., b_{n_x}), where b_j is a gene represented as a bit vector of length n_d. The total number of bits in the bitstring (n_b) is: $n_b = n_x n_d$. b_j can be decoded to give f_j using the following[14]

$$f_j = x_{min,j} + \frac{x_{max,j} - x_{min,j}}{2^{n_d} - 1}\left(\sum_{l=1}^{n_d-1} b_{j(n_d-l)} 2^l\right) \tag{2.2}$$

x_{min} and x_{max} are the lower and upper bounds of the search space, meaning that all decoded values will lie in the range [x_{min}, x_{max}]. As such, GAs are inherently constrained and do not need constraint handling mechanisms (discussed in a later section) to prevent exploration outside the search space.

By rearranging equation 2.2, the equation for encoding individuals is obtained:

$$b_j = round\left(\frac{f_j - x_{min,j}}{\frac{x_{max,j} - x_{min,j}}{2^{n_d}-1}}\right) \tag{2.3}$$

A slight variation on binary encoding is grey coding. The problem with traditional binary coding is that it introduces *Hamming cliffs*. "A Hamming cliff is formed when two numerically adjacent values have bit representations that are far apart"[14]. For example, the number 7 in 4-bit binary is 0111 and the number 8 is 1000. In order to move from 7 to 8, the values of all 4 bits have to change; there is a *Hamming distance* of 4. This can cause problems for a GA, as a larger number of mutations are required to get to 8. Grey coding resolves this problem by ensuring that the Hamming distance between successive numbers is always 1, see table 2.1. Binary code is converted into grey code as follows[14]:

$$\begin{aligned} g_1 &= b_1 \\ g_l &= b_{l-1}\bar{b}_l + \bar{b}_{l-1} b_l \end{aligned} \tag{2.4}$$

where b_l is the l^{th} bit in the binary code, b_1 is the most significant bit, $\bar{b}_l$ means not b_l, + means logical OR and multiplication means logical AND. Grey code values are converted into the phenotype using a modified version of equation 2.2[14]

$$f_j = x_{min,j} + \frac{x_{max,j} - x_{min,j}}{2^{n_d} - 1}\left(\sum_{l=1}^{n_d-1} \left(\sum_{q=1}^{n_d-l} b_{(j-1)n_d+q}\right) mod\ 2\right) 2^l \tag{2.5}$$

Real value encoded GAs have also been proposed. The use of real values makes recombination

Value	Binary Code	Grey Code
0	0000	0000
1	0001	0001
2	0010	0011
3	0011	0010
4	0100	0110
5	0101	0111
6	0110	0101
7	0111	0100
8	1000	1100
9	1001	1101
10	1010	1111
11	1011	1110
12	1100	1010
13	1101	1011
14	1110	1001
15	1111	1000

Table 2.1: Differences between binary and grey coding

and mutation more complicated, whilst making evaluation simpler. For this project, only binary or grey coded GAs were considered.

2.3.5.2 Selection

Selection operators in GAs are used for two purposes, first to select individuals for use in recombination (mate selection) and second to select the population of the next generation (reinsertion or environmental selection). There are many selection methods and the canonical GA makes use of proportional selection.

Proportional selection uses a probability distribution which is sampled to select individuals from the population. This selection method is biased towards the most fit individuals in the population but is not an elitist selection method. Elitist selection methods guarantee that the most fit individual(s) in a population will be selected. With proportional selection, whilst the most fit individuals have a higher probability of selection, it is possible that they may not be selected. The probability of selection for each individual is calculated using its fitness value[15–18]

$$P(x_i) = \frac{f(x_i)}{\sum_{j=1}^{n} f(x_j)} \tag{2.6}$$

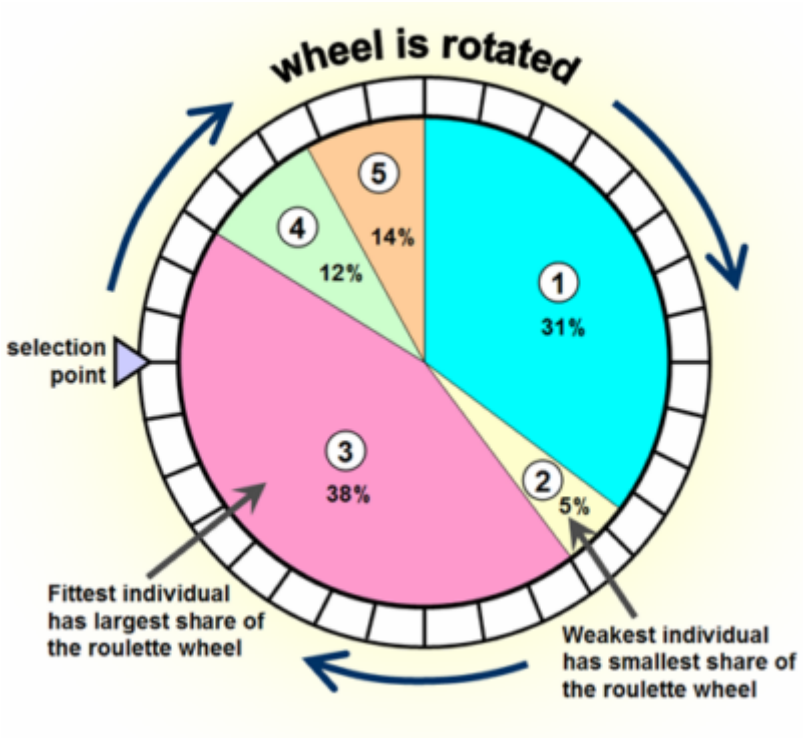


Figure 2.4: Representation of roulette wheel selection

Once the probability of selection for each individual has been calculated and combined into a probability distribution, you can sample it and select as many individuals as necessary.

Proportional selection is also known as roulette wheel selection. Sampling the probability distribution can be thought of as spinning a roulette wheel, where each individual has been assigned a portion of the wheel proportional to their selection probability (see fig 2.4[19]).

Proportional selection as defined here will not work for minimisation problems, as the most fit individuals will be assigned a small probability of selection. In order to use proportional selection for minimisation problems, the fitness values of the population need to be scaled. One method of scaling is[15]

$$scaledfitness(x_i) = f_{max} - f(x_i) \tag{2.7}$$

with f_{max} as the maximum possible fitness value. If the maximum possible fitness is not known, then the maximum observed fitness, $f_{max}(t)$, can be used. Another scaling method is[15]

$$scaledfitness(x_i) = \frac{1}{1 + f(x_i) - f_{min}(t)} \tag{2.8}$$

The advantage of proportional selection is that it preserves diversity by affording all individuals a probability of selection[16, 18]. However, due to the direct relationship between the fitness of the individuals and the selection probabilities, it is possible for a small number of very fit

individuals to dominate selection, potentially resulting in premature convergence[15, 18].

Rank based selection is another selection method. This is similar to proportional selection, except that instead of using the fitness values to directly calculate selection probabilities, the fitness values are used to assign the individuals a rank. The population is first sorted according to the fitness values, and then the individuals are assigned a rank, either so the best individual has rank n (where n is the population size)[17] or rank 0[15]. The rank of the individual is then used to calculate the selection probability. The resulting probability distribution can then be sampled to select individuals. Methods to assign selection probabilities include the linear ranking method

$$\begin{aligned} &P(x_i) = \frac{1}{N}(\eta^- + (\eta^+ - \eta^-)\frac{i-1}{N-1})[17] \\ &or \\ &P(x_i) = \frac{1}{N}(\eta_{max} - 2(\eta_{max} - 1)\frac{i-1}{N-1}[16] \end{aligned} \tag{2.9}$$

For this project however, the following method is used[20]:

$$P(x_i) = \frac{rank(x_i)}{\sum_{j=1}^{n} rank(x_j)} \tag{2.10}$$

This method was chosen due to its simplicity and the previous experience of the author.
Rank based selection attempts to deal with the disadvantage of proportional selection; that a few very fit individuals are able to dominate the selection. Because the fitness values are not used directly in calculating the selection probability this problem is avoided. It does not matter how much more fit the best individual is compared with the others, it will only be one rank higher. Rank based selection has its own disadvantages though, it can be computationally expensive, due to the need to sort the population, and may still lead to premature convergence as the best chromosomes are not so strongly differentiated[18].

A third selection method is tournament selection. This is a commonly used method due to its simplicity and efficiency. A number of individuals are randomly selected from the population, and the best individual is selected[15–18]. The number of individuals selected from the population each time is called the tournament size. Careful selection of the tournament size is important in order to prevent the best or worst individuals from dominating, leading to a reduction in diversity[15, 18]. A common tournament size is 2, and tournaments of this variety

```
offspring1 = parent1, offspring2 = parent2;
Compute bit mask m;
for i = 1, ..., n do
    if m[i] = 1 then
        offspring1[i] = parent2[i];
        offspring2[i] = parent1[i];
    end
end
```

Algorithm 1: Using a bit mask for crossover

are called *binary tournaments*[17, 18].

For details on other selection methods, see [15] and [17].

2.3.5.3 Recombination

The recombination operator for GAs is called crossover. When dealing with binary or grey coded GAs, crossover generally involves swapping sections of the bitstrings between two parent individuals. This is an example of *sexual* crossover, where two parents individuals are used to produce one or two offspring. *Asexual*; one parent producing one offspring, and *multi-recombination*; more than two parents producing one or more offspring, crossover variants also exist but for this project only sexual methods were investigated.

A bit mask, the same length as the bitstring, is used to control which bits are exchanged between the parent individuals. The bit mask is read a bit at a time, and if the bit is set, the corresponding bits in the parents are exchanged to produce offspring (see algorithm 1[11]). The purpose of a crossover operator is to compute the bit mask. Three operators have been developed for binary crossover, fig 2.5(a)[21] shows a summary of the three.

- One-point crossover. One point in the bitstring is randomly selected, this is called the crossover point. The bit mask is computed such that all bits after the crossover point are set. Used by the canonical GA
- Two-point crossover. Two points in the bitstring are randomly selected; a start and end point. The bit mask is computed such that all bits between these two points are set. Fig 2.5(b)[20] shows an example of two-point crossover with the bit mask

- Uniform crossover. A random bit mask is generated. Each bit in the mask has a probability of being set. Fig 2.5(c)[20] shows an example of uniform crossover with the bit mask

A configurable parameter controls crossover. Called the *crossover rate*, it is effectively the probability that crossover will take place. Generally, a high crossover rate is used; between 0.5 and 1[11, 22].

2.3.5.4 Mutation

Mutation is applied to the offspring produced by crossover in order to add further genetic variation to the population. In binary or grey coded GAs, the process is very simple; the value of certain bits is inverted. The bits to invert are selected using one of two methods

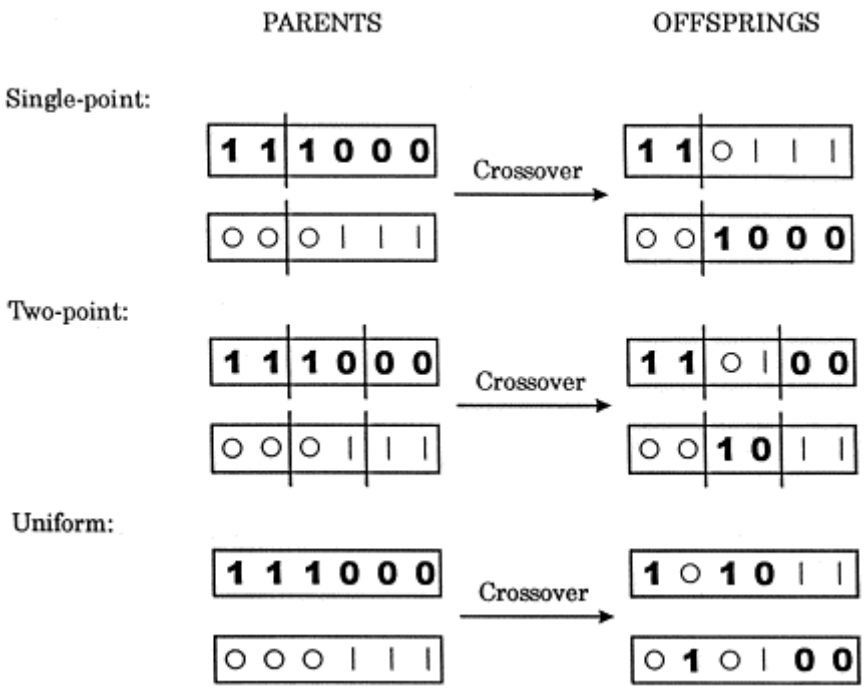
- Uniform mutation. Bits are chosen at random and inverted. Used by the canonical GA
- Inorder mutation. Similar to two-point crossover, a start and end point is randomly selected and all bits between these points are inverted.

Gaussian mutation similar to that used in ES (see section 2.3.6.4) can be used for GAs where the phenotype is floating-point numbers. The genotype is first decoded into the phenotype, gaussian mutation is applied and then the result is converted back into the genotype.[11]
As with crossover, mutation is controlled by a parameter, called *mutation rate*. This is the probability with which the selected bits are inverted. Typically a small value is used for the mutation rate[11, 22].

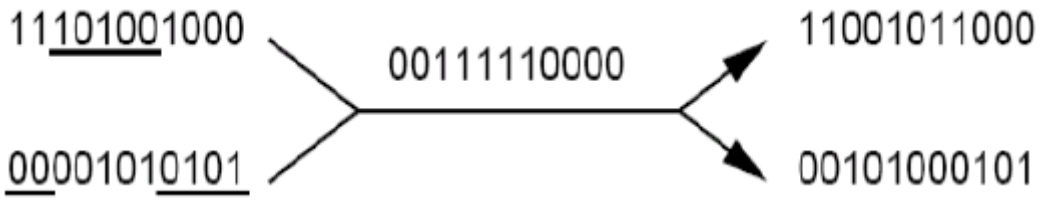
2.3.5.5 Configurable Parameters

A GA has a number of configurable parameters that can affect the performance and output of the algorithm. Careful selection of these parameters is essential to optimise the performance of the GA, and while they may be problem dependent and number of standards have been established[23, 24].
The key parameters are

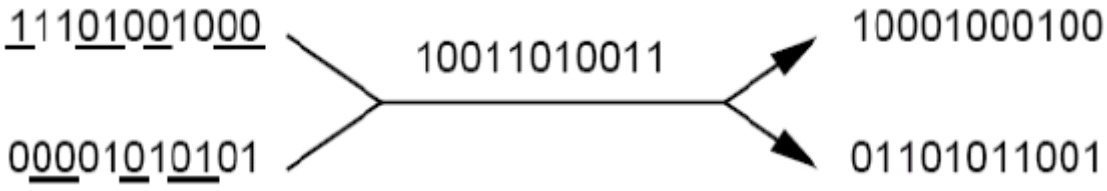
- Population size
- Number of generations



(a) Crossover operators



(b) Two-point crossover with bit mask



(c) Uniform crossover with bit mask

Figure 2.5: Summary showing the three sexual crossover operators (a) with bit masks (b), (c)

- Number of bits to represent each gene
- Selection method and tournament size (if appropriate)
- Crossover type and rate. Crossover is explorative, meaning it is used to discover good search areas; *exploring* the search space. So the crossover rate needs to be high to allow the algorithm to find the global optimum
- Mutation type and rate. Mutation is exploitative, meaning it explores locally within an area. So the mutation rate should be small in order to reduce the size of the area searched and prevent good solutions being changed too much. If the mutation rate is too small however, very little exploration will take place and the population may converge to a local optimum

2.3.6 Evolution Strategies

First developed in the 1960s by Rechenberg and later by Schwefel, Evolution Strategies (ES) are based on the idea of "the evolution of evolution"[25]. Unlike GAs which are designed to replicate evolution on a genetic level, ES are designed on species-level evolution; the emphasis is towards the phenotype rather than the genotype.
The first ES suggested was the (1 + 1)-ES and does not use a population; one individual is used and one offspring is produced using a mutation operator[25, 26]. The first multimembered ES, (μ + 1), was developed by Rechenberg. It uses μ parents, selects 2 randomly and produces one offspring from them and the best μ individuals are selected into the population[25, 26]. Schwefel later introduced two more multimembered ES: (μ, λ) and (μ + λ)[26], which use a population of μ parents and produce λ offspring. These types of ES are often shown as ($\mu \overset{+}{,} \lambda$).

2.3.6.1 Encoding

ES individuals are encoded using real value coding, which better reflects the focus on the phenotype of the individuals. There is therefore no need to decode the individuals before fitness evaluation, as the values of the decision variables are stored directly.
In addition to the set of decision variables, individuals in ES are also coded with a set of strategy parameters. The strategy parameters are used to control the mutation operator and influence the search direction and step size[25, 26]. The use of strategy parameters means that an ES

always moves towards the optimum which give ES an advantage over GAs, which can move in any direction. As such, it is expected that ES have a faster convergence speed than GAs.
It is incredibly important to adjust the value of the strategy parameters throughout the evolution process, otherwise the ES loses it's evolvability[26]. The traditional way of mutating the strategy parameters was using the 1/5th-rule[26]. However, this restricts the number of strategy parameters to one and is really only applicable to $(1 + 1)$-ES. A better way to adjust the strategy parameters is *self-adaptation*. Self-adaptation means that the strategy parameters are evolved at the same time as the decision variables; the strategy parameters undergo recombination and are mutated along with the decision variables.

2.3.6.2 Selection

Selection in ES is much simpler than in GAs; the μ best individuals are selected from a pool of individuals. How the pool of individuals is populated depends on the type of ES.

With $(\mu + \lambda)$-ES, the pool of individuals is the combination of the μ parents and the λ offspring. This method places no limits on the number of offspring, since you can always guarantee that at least μ individuals will be in the selection pool. This is an elitist selection method; it always ensures that the most fit individuals will survive into the next generation[25–27]. It is possible with this selection method for parent individuals to survive in the population indefinitely. In order to prevent this, $(\mu + \lambda)$ can be enhanced to (μ, k, λ), where k is the lifespan of an individual; individuals will be discarded if they exceed their lifespan[25].

For (μ, λ)-ES, the selection pool is the λ offspring only, the parent individuals are discarded. Therefore, this selection method requires that $\mu < \lambda$. This selection method is not elitist, as the parents are discarded, and therefore results in greater diversity within the population[25–27].

2.3.6.3 Recombination

As opposed to GA recombination, ES recombination produces only one offspring from ρ parents, where $2 \leq \rho \leq \mu$. Recombination where $\rho = 2$ is called local recombination and when $\rho > 2$ is called global[25], or multi[26], recombination. There are two simple methods of recombination, *discrete* and *intermediate*.

Discrete recombination is denoted as $(\mu/\rho_D \dagger \lambda)$. In this method, each decision variable in the offspring corresponds directly to the decision variable in a randomly selected parent. For a parent decision vector $a = (a_1, ..., a_n)$ and an offspring $r = (r_1, ..., r_n)$,[26]

$$\begin{aligned} r_k &= (a_p)_k \\ &\text{where } p = random[1, ..., \rho] \end{aligned} \tag{2.11}$$

So, the k^{th} component of r is the k^{th} component of the p^{th} parental vector, a, with p as a random parent.

Intermediate recombination is denoted as $(\mu/\rho_I \dagger \lambda)$ and uses the average value of the decision variables in the parents to produce the offspring. For a parent decision vector $a = (a_1, ..., a_n)$ and an offspring $r = (r_1, ..., r_n)$,[26]

$$r_k = \frac{1}{\rho} \sum_{p=1}^{\rho} (a_p)_k \tag{2.12}$$

So, the k^{th} component of r is the average value of the k^{th} component in all the parental vectors, a.

2.3.6.4 Mutation

Mutation in ES happens with a probability of 1, as opposed to a very low probability in GAs. Mutation takes place in two steps, first mutation (self-adaptation) of the strategy parameters and then mutation of the decision variables.

When there is only one strategy parameter, an offspring y is mutated to y' as follows[25, 26]

$$\begin{aligned} y' &= y + z \\ &\text{where } z = \sigma(N_1(0,1), ..., N_n(0,1)) \end{aligned} \tag{2.13}$$

Each decision variable of y is mutated by $\sigma(N(0,1))$, where σ is the strategy parameter and $N(0,1)$ is a random sample from a Gaussian (normal) distribution; hence why ES mutation is called Gaussian mutation.

This can be modified to support multiple strategy parameters[25, 26]

$$\begin{aligned} y' &= y + z \\ &\text{where } z = (\sigma_1 N_1(0,1), ..., \sigma_n N_n(0,1)) \end{aligned} \tag{2.14}$$

Now, a separate strategy parameter is used to mutate each decision variable in the offspring. Obviously, this now requires that the number of strategy parameters and decision variables are

the same; $n_s = n_d$.

The most common method of mutating strategy parameters is the log-normal method, so called because of the logarithmic normal distribution it generates

A strategy parameter, σ, is mutated to σ' using the log-normal method as follows[25, 26]

$$\sigma' = \sigma exp(\tau N(0,1))$$

where

$$\tau = \frac{1}{\sqrt{n}} \tag{2.15}$$

n is the size of the search space, i.e. the number of decision variables

This method of mutation works for one strategy parameters, and is slightly modified if more than one strategy parameter needs mutating.

A strategy parameter, σ_i, in a set $\sigma = (\sigma_1, ...\sigma_n)$, is mutated to σ'_i as follows[25, 26]

$$\sigma'_i = \sigma_i exp(\tau' N(0,1)) exp(\tau N_1(0,1))$$

where

$$\tau' = \frac{1}{\sqrt{2n}} \tag{2.16}$$
$$\tau = \frac{1}{\sqrt{2\sqrt{n}}}$$

n is the size of the search space, i.e. the number of decision variables

2.3.6.5 Constraint Handling

Unlike GAs, ES are not inherently bound to a finite search space. Unconstrained, an ES will explore outside the bounds of a search space to find the optimum. If the algorithm is required to only search a finite space then a method of constraint handling must be implemented. Constraint handling methods are also implemented to handle any other constraints that there may be on the function being optimised, so the techniques listed here are also applicable to GAs in that sense. Constrained optimisation problems split the search space into feasible and infeasible regions. Figure 2.6[28] shows the feasible and infeasible regions for some function. It might be the case that to reach the optimum the algorithm needs to pass through an infeasible region, so constraint handling mechanisms should still allow infeasible solutions to be selected. For example, in fig 2.6, if the optimum was at d, the algorithm would have to move through the infeasible region to reach it.

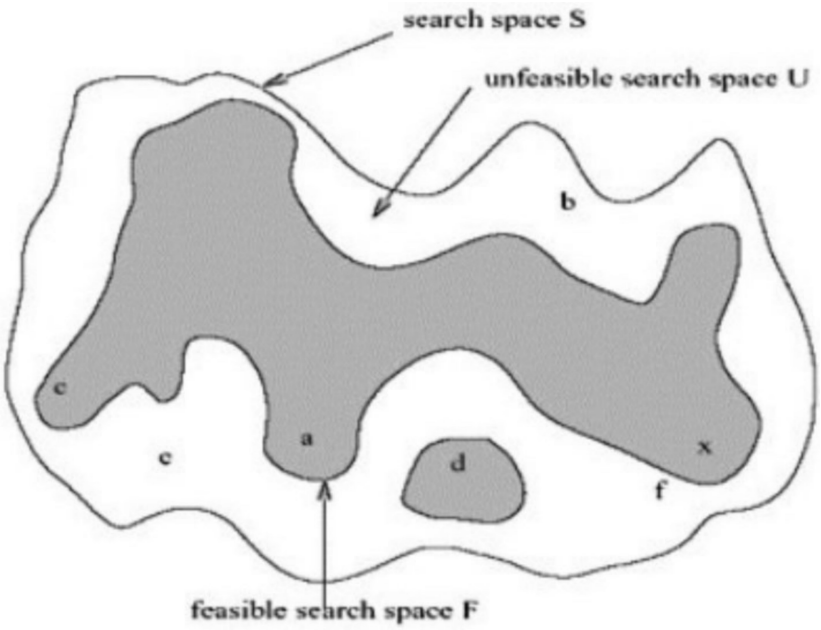


Figure 2.6: A search space, showing the feasible and infeasible regions

One method of constraint handling are *penalty functions*. With penalty functions, the constrained problem is converted into an unconstrained problem by adding a term to the fitness function which is "based on the amount of constraint violation"[29]. For a constrained optimisation problem defined as in equation 2.1, a penalised fitness function is usually takes this form[11, 29]

$$f'(x) = f(x) \pm (\sum_{i=1}^{n} r_i G_i + \sum_{j=1}^{p} c_j L_j)$$

where

G_i is a function of the constraints $g(x)$, typically $max[0, g_i(x)]^\beta$

L_j is a function of the constraints $h(x)$, typically $|h_j(x)|^\gamma$

β and γ are usually 1 or 2

Equality constraints $h(x) = 0$ are converted to inequality constraints $|h(x)| - \varepsilon \leq 0$

ε is the tolerance allowed (small)

r_i and c_j are constants called penalty factors

(2.17)

There are various methods for defining the value(s) of the penalty factors.

Static penalty factors are one approach, and use constant penalty factors. One method, suggested by Homaifar, Lai and QI in 1994, defines multi- levelled penalty factors based on the level of constraint violation. Higher levels of violation have higher penalty factors. The penalised fitness if calculated as follows[11, 28, 29]

$$f'(x) = f(x) + (\sum_{i=1}^{n} R_{j,i} G_i) \tag{2.18}$$

where $R_{j,i}$ is the j^{th} violation level for constraint i

This method has some disadvantages, primarily that a large number of parameters need to be created; each constraint has a number of violation levels that need to be created and penalty factors may be problem dependent[28, 29].

Dynamic penalty factors are another approach, and involves the generation number in setting the penalty factors. Using this approach, the penalty factors increase over time. Joines and Houck suggest a method where the penalised fitness is calculated by[11, 27–29]

$$f'(x) = f(x) + (C.t)^{\alpha} (\sum_{i=1}^{n} G_i^{\beta}(x)) \tag{2.19}$$

where C, α, β are user defined constants and t is the generation number

typically $C = 0.5$, $\alpha = 1$ or 2 and $\beta = 1$ or 2

Experiments have shown that the values of C, α and β can seriously affect the performance of an EA, potentially leading to premature convergence or convergence to a local optimum[28]

Another method of constraint handling is to simply reject infeasible individuals. This is often called the *death penalty* method. This approach is very simple, infeasible solutions are discarded and new solutions generated until there are enough feasible solutions. This method works reasonably well with large feasible search spaces but has been shown, by experiment, to perform badly with small feasible regions[27–29].

Repair methods are a third constraint handling mechanism. These methods make use of a repair algorithm which "generates a feasible solution from an infeasible one"[30]. The repaired solution can then be used to evaluate its infeasible equivalent, or it can replace the infeasible solution in the population[28, 30]. The specifics of the repair algorithm are usually problem

dependent, so there are few standard repair techniques[28].
Many other constraint handling mechanisms exist, see [28] for details.

2.3.6.6 Configurable Parameters

ES have similar configuration parameters to GAs.
The key parameters are

- Population size
- Number of generations
- Number of strategy parameters
- Selection method
- Number of offspring to produce. Must be greater than μ if using (μ, λ) selection
- Number of parents to use to produce offspring
- Recombination method

2.4 Swarm Intelligence

A *swarm* can be defined as "a group of (generally mobile) agents that communicate with each other (either directly or indirectly), by acting on their local environment"[31]. Swarm intelligence is the complex behaviour and problem-solving abilities developed by a swarm. Individuals in a swarm are simple, so the complex behaviour swarms exhibit come from the interactions between individuals, as opposed to the individuals themselves. This is called *emergence*[31]. Swarms of bees, ant colonies and bird flocks are natural examples of intelligent swarms and swarm intelligence algorithms aim to simulate this.
Whilst many swarm intelligence algorithms exist, this report is concerned with only one, Particle Swarm Optimisation (PSO).

2.4.1 Particle Swarm Optimisation

Developed in the 1990s by James Kennedy and Russell Eberhart, PSO attempts to model the behaviour of birds in a flock[32, 33]. The complex behaviour displayed by PSO is that of "all

individuals converging on the environment state that is best for all individuals"[31], i.e. the global optimum. It can be thought that individuals, or particles, in the population are 'flown' around the solution space "according to its own experience and that of its neighbours"[32, 33].

2.4.1.1 Basic Particle Swarm Optimisation Algorithm

PSO is much simpler than EAs, discussed in section 2.3. There are no complex encoding methods or genetic operators. Particles are represented as a position (the decision variables) and velocity vector and each iteration the position and velocity of each particle is updated. The position of particle i at time t, $x_i(t)$, is updated using the velocity of the particle, $v_i(t)$[32–34]

$$x_i(t+1) = x_i(t) + v_i(t+1) \tag{2.20}$$

The velocity of the particles reflects both the experience of the particle (*cognitive component*) and the experience of neighbouring particles (*social component*). The size of the particle's neighbourhood can vary, and this gives rise to different PSO variants. The original two variants are *gbest* and *lbest*.

2.4.1.2 gbest Particle Swarm Optimisation

In a gbest, or *global best*, PSO, a particle's neighbourhood is the entire swarm. This means the social component of each particle's velocity is taken from all the particles. In this case, the social component is the best value found by any particle in the swarm (*global best value*). Hence, the velocity for a particle is updated as follows[32, 34]

$$v_i(t+1) = v_i(t) + c_1 r_1 (pbest_i(t) - x_i(t)) + c_2 r_2 (gbest(t) - x_i(t))$$

where

c_1 and c_2 are user supplied acceleration constants, typically set to 2[34, 35] (2.21)

r_1 and r_2 are fresh uniform random numbers

$pbest_i(t)$ is the *personal best value*, i.e. the best value seen, of individual i

This gives us the pseudocode for a gbest PSO, see algorithm 2.

2.4.1.3 lbest PSO

A lbest, or *local best*, PSO, uses a neighbourhood of the k nearest neighbours of a particle. The social component of a particle's velocity is still the best value found by the neighbourhood, but

```
initialise a population;
while stopping criteria not met do
    update global best value;
    for all particles do
        update personal best value;
        update velocity vector using equation 2.21;
        update position vector using equation 2.20;
    end
end
```

Algorithm 2: gbest Particle Swarm Optimisation algorithm

```
initialise a population;
while stopping criteria not met do
    for all particles do
        update local and personal best values; update velocity vector using equation 2.22;
        update position vector using equation 2.20;
    end
end
```

Algorithm 3: lbest Particle Swarm Optimisation algorithm

a small neighbourhood is used now; the social component is called the *local best value*. The velocity for a particle is updated by[32]

$$v_i(t+1) = v_i(t) + c_1 r_1 (pbest_i(t) - x_i(t)) + c_2 r_2 (lbest_i(t) - x_i(t))$$

where

$$\begin{aligned} &c_1 \text{ and } c_2 \text{ are user supplied acceleration constants, typically set to 2[34, 35]} \\ &r_1 \text{ and } r_2 \text{ are fresh uniform random numbers} \\ &pbest_i(t) \text{ is the } \textit{personal best value}\text{, i.e. the best value seen, of individual } i \\ &lbest_i(t) \text{ is the local best value of the neighbourhood of individual } i \end{aligned} \tag{2.22}$$

This gives us the pseudocode for a lbest PSO, see algorithm 3.

2.4.1.4 Inertia Weight

Another variant on the velocity update equation is to add an inertia weight component. The inertia weight controls the influence the previous velocity has on the new velocity[32, 33, 35].

Taking the gbest velocity equation (equation 2.21), adding an inertia weight changes the equation to the following

$$v_i(t+1) = \omega v_i(t) + c_1 r_1(pbest_i(t) - x_i(t)) + c_2 r_2(lbest_i(t) - x_i(t)) \quad (2.23)$$

where ω is the inertia weight component

The conventional use of inertia weight components is as a static constant, typically defined between 0.8 and 1.2[34]. A better way to use inertia weights is as a varying weight.

A large inertia weight encourages the PSO to be more explorative and a small inertia weight makes the PSO more exploitative. As such, it makes sense to decrease the value of the inertia weight over time; called a *time-varying inertia weight*. This will allow the particles to widely explore the search space at the start of the algorithm, preventing it getting stuck at local optima, and then explore with greater detail in a local area, ensuring that the global optimum will be found.

A time-varying inertia weight is usually linearly decreased at each iteration, from a maximum value, w_{max} to a minimum value, w_{min}. w_{max} is usually set to 0.9 and w_{min} to 0.4[32, 35]. The inertia weight is updated using the following equation[32, 35, 36]

$$w(t) = \frac{(w_{max} - w_{min})(t_{max} - t)}{t_{max}} + w_{min} \quad (2.24)$$

where t_{max} is the maximum number of iterations

There are other methods of adjusting the inertia weight, see [32] for details.

2.4.1.5 Velocity Clamping

Velocity clamping is used to prevent velocity explosion; where the velocity of the particles rapidly increases to large values, meaning the positions of the particles are updated in large jumps. Essentially, a PSO with velocity explosion is too explorative; the particles explore outside the bounds of the search space. To prevent this, a new parameter is introduced, v_{max}. This sets a maximum limit on velocity of a particle and if the updated velocity exceeds this value, it is set to v_{max}[32, 34, 35]. In an optimisation problem bounded by $[-x_{max}, x_{max}]$, the value of v_{max} is[34]

$$v_{max} = kx_{max} \quad (2.25)$$

where k is a user defined constant, the *velocity clamping factor*

If the problem is not bounded by $[-x_{max}, x_{max}]$, but instead by $[x_{min}, x_{max}]$, then the value of v_{max} is determined by[32, 34]

$$v_{max} = \frac{k(x_{max} - x_{min})}{2} \tag{2.26}$$

The value k has been shown to be problem dependent and should be defined in the range $(0,\ 1]$[32, 34].

For particles with more than one search dimension, a separate v_{max} should be used for each dimension, using the corresponding x_{min} and x_{max}.

One problem with velocity clamping is when all the velocities are equal to v_{max}. In this case, it may prevent the PSO from finding the optimum. The use of an inertia weight can help to prevent this. Another solution is to dynamically decrease v_{max} over time, starting with a large value to allow exploration; dynamic velocity clamping is not discussed in this report, see [32].

2.4.1.6 Bound Handling

When dealing with constrained optimisation problems, many of the constraint handling methods used in EAs, see section 2.3.6.5, are appropriate. It is important that the way the personal best and neighbourhood best values are updated handles infeasible solutions too, i.e. the personal or neighbourhood best should only be changed for particles in feasible regions[32]. Several methods have been defined for dealing with variable bounds (called *bound handling*) in PSO, and some of which will be discussed briefly here.

The simplest bound handling method is the random method. The particle's position vector is checked sequentially, and if a variable exceeds its bounds it is moved to a random location between x_{min} and x_{max}. This method can have a great impact on convergence accuracy due its random nature[37].

The set on boundary method is another simple bound handling method. Using this method, variables that exceed their bounds are set to the value of the bound they exceed, e.g. if x_{max} is exceeded, the variable will be set to x_{max}. A variant of this method makes use of *velocity reflection*; if a particle's i^{th} variable is set to the boundary, then the i^{th} velocity will by reflected such that the particle is moving in the opposite direction[37].

A third bound handling method is exponential distribution. This approach adjusts the position of any infeasible variable to a point between its original position and the bound it exceeded. The position of the variable is sampled from a probability distribution calculated such that positions near the bound have a higher selection probability[37].
More details on these bound handling methods and a few others can be found in [37].

2.4.1.7 Configurable Parameters

PSO has very few configurable parameters, and the majority are the same as ES and GAs
The key parameters are

- Population size
- Number of generations
- Velocity clamping factor, k, if using velocity clamping

2.5 Connect 4

Connect 4 is a 2-player game where each player has the same number of identical counters in different colours; typically red and yellow. The goal of each player is to connect four of their counters in either a vertical, horizontal or diagonal line, the first player to do so is the winner. If all the counters for both players have been played without connecting four in a line, then the game is a draw. The game is played on a grid of varying size, but usually 7 columns by 6 rows, and is gravity based, meaning that when a counter is placed into a column it will fall to the lowest unoccupied row in that column.

Connect 4 was solved in 1988 by Victor Allis and James D. Allen independently. In his master's thesis, Allis describes 9 strategic rules by which to play Connect 4. These rules are based on controlling the *Zugzwang*, a concept Allis defines as forcing "a player to make a move which he would rather not make"[1]. The rules Allis defines are listed below. An explanation of the rules has been left out, due to their the relative complexity; interested readers should see [1].

- Claimeven

- Baseinverse
- Vertical
- Aftereven
- Lowinverse
- Highinverse
- Baseclaim
- Before
- Specialbefore

Allis also describes a program, VICTOR, which plays Connect 4 using these rules. Using VICTOR, Allis proved that, given perfect play, the first player will always win if they play the first move in the middle column, will always lose if they play the first move in columns 1 and 7, and will always draw if they play the first move in any other column[1].

2.6 Technologies

2.6.1 The Arena

The Arena is a Java based framework developed to allow player "modules" to play against each other automatically. The Arena provides a number of games that modules can be developed to play, but the primary game is Connect 4. It also provides a number of predefined modules, with which to test new modules. New modules can be created by simply implementing an interface, and providing an implementation of a function which returns the next move to play in the game. This function is called each turn by The Arena, and the move returned is played. Each turn is restricted by a time limit, and the opponent will win automatically if the time limit is exceeded by a player. This allows the effectiveness of modules, such as the recursive module, which analyse the future states of the board to find the best move, to be limited; as only so many states can be analysed in the time limit. The Arena also provides a method of storing a move which will be automatically played if the turn time limit runs out; allowing modules like the recursive module to continue to analyse the board until the very end of their turn.

Library	Supported Algorithms
ECJ [38]	GAs, ES, PSO + more
cilib [39]	GAs, ES, PSO + more
JGAL [40]	GAs
JGAP [41]	GAs
JSwarm-PSO [42]	PSO

Table 2.2: Java computational intelligence libraries

Once a new module is written it simply needs to be published as a jar file and then it can be loaded into The Arena.

The Arena provides both a command line and graphical user interface (GUI). The GUI provides a graphical representation of the board as well as menus to load player modules and set the turn time limit; it is more suited to a human player playing against the computer. The command line interface is much more powerful. It allows tournaments to be set up, where player modules are played against each other over a number of rounds and allows a machine to be set up as a host for a tournament, allowing the games to be run on multiple slave machines, reporting the results back to the host via TCP.

Since The Arena is Java based, the main development language for this project was Java.

2.6.2 Computational Intelligence Libraries

There are a number of libraries providing CI algorithms, some of which are listed in table 2.2. However, due to the project's aims and the author's goals, the use of a library was not considered for the main body of this project, and the algorithms chosen where implemented from scratch. However, as part of the last aim of the project, it would be helpful to compare the performance of the produced algorithms against those implemented by a popular CI library. As table 2.2 shows, there are a number of appropriate Java libraries. However, due to the author's previous experience and time limitations, the Shark machine learning library[43] was chosen. Shark release 2.3.4[44] was used as at the time of development, this was the most up to date release build available.

As Shark is a C++ based library, and this project was developed in Java, the performance comparisons could not be against run speed, as C++ (a compiled language) has an advantage over Java (an interpreted language). Instead, the convergence speed and accuracy of the algorithms

would be compared.

Chapter 3

System Specification

3.1 System Requirements

Before proceeding with the design and implementation of any software project it is important to define the requirements of the system to be developed. The requirements of a system define what the system must be able to do and are normally split into functional and non-functional requirements. Functional requirements of a system specify the explicit behaviour of the system and non-functional requirements specify other properties of the system that aren't explicitly related to the functionality, e.g. performance, file types supported etc.
In a professional software engineering environment the requirements gathering process would be conducted with the end user of the system, in order to ensure that what is developed matches what the end users actually want, and not what the developers think they might want. In this situation, a variety of requirement gathering techniques would be employed, some of which may include

- Document analysis; analysing any existing documents the end users currently use. This can help to establish the structure of database tables, or simply the datatypes the system should use
- Interviews. By interviewing end users, it is easy to get a list of what they want as well as an idea on how the system currently works
- Observation. By observing the current system, an idea of the flow of data etc. can be gathered

- Research. Research into the technologies available, or other similar solutions, can be a key factor in specifying a system

Due to the nature of the project there is no specific end user, so the requirements were based on the research described in the previous section.

3.1.1 Functional Requirements

- The system must be able to play Connect 4 in The Arena as effectively as possible
- The system must be able to optimise any unconstrained real valued maximisation or minimisation problem
- The system must optimise within the bounds defined by the problem only
- The system must implement the genetic, evolution strategies and particle swarm optimisation algorithms as described in the previous section
- The genetic and evolution strategies algorithms must be implemented supporting a number of the different selection and recombination methods described in the previous section
- The state of the system must be able to be written to and loaded from a file
- The system must store the fitness of each individual at each generation, and must allow this data to be plotted on a scatter graph

3.1.2 Non-functional Requirements

- The parameters of the algorithms must be easily changeable
- The system must allow new problems to be defined as easily as possible
- The system must allow problem specific stopping criteria to be easily defined
- The system must allow new binary or real value coded population based algorithms to be added easily
- The system must be fully documented using JavaDoc
- The system must be machine and operating system independent

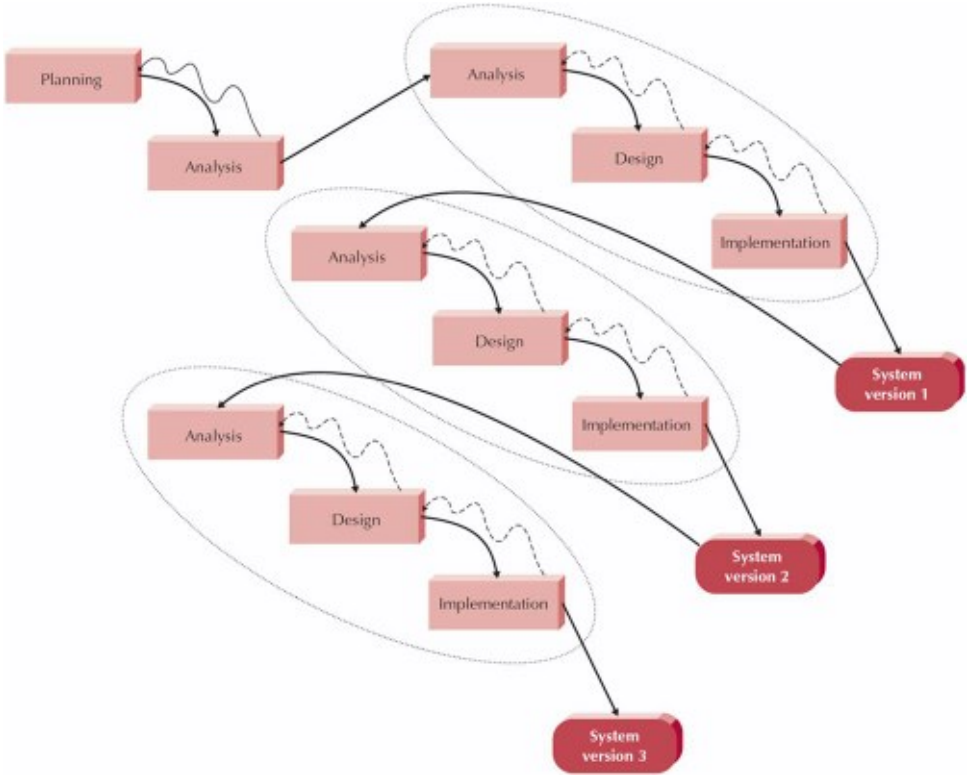


Figure 3.1: Phased development methodology

3.2 Development Methodology

Using a software development methodology helps to keep any large scale development project in check. Development methodologies split development into a number of phases, which build on top of each other and guide the development of the software. Typical phases in a development methodology are analysis, design, implementation and testing.

The primary development methodology chosen for this project was phased development. In this methodology, the system is developed in phases of design, implementation and testing, with each phase building on the previous one, as shown by figure 3.1[45]. The use of this methodology is supported by the project's first and second aims. These aims can be used to split the project into two developmental phases; first, to develop a generic optimisation library and second to adapt this library to The Arena.

Figure 3.2: Gantt chart showing the development timescale

3.2.1 Development Timescale

Figure 3.2 shows the gantt chart of the planned development timescale. Following the end of phase two testing there is a testing phase during which the algorithms will be run, attempting to learn how to play Connect 4. As large period of time as possible, 4 weeks, was allocated for this, as it was unknown how long training would take when planning the project.

Chapter 4

System Design

Once the requirements of the system had been defined, and a development methodology chosen, the design process could begin. As a phased development methodology was chosen, with two phases, the design process was split into two sections.

4.1 Phase One: Generic Optimisation Library

The first phase of development was the generic optimisation library. As defined by the system requirements, this had to be capable of maximising or minimising any unconstrained real value problem and had to implement the genetic, evolution strategies and particle swarm optimisation algorithms with a number of selection and recombination modes.
The first step in designing the library was to select which selection and recombination modes should be implemented.
For the GA, the following selection and recombination methods were chosen

- Proportional selection
- Rank selection
- Binary tournament selection
- One point crossover
- Two point crossover

These three selection methods were chosen because of their simplicity yet effectiveness. Originally, the GA was implemented with only rank and proportional selection. Binary tournament selection was added later after some initial testing. Only binary tournament support was added, instead of a variable size tournament, in order to avoid adding another configurable parameter which could affect the performance of the GA.
ES was implemented with all the selection and recombination methods discussed, so

- (μ, λ) selection
- $(\mu + \lambda)$ selection
- Discrete recombination
- Global discrete recombination
- Intermediate recombination
- Global intermediate recombination

The ES algorithm was also implemented using the repair methodology to provide constraint handling. As the requirements of the project specify that the system must be able to deal with any unconstrained problem but must also only optimise within the problem's bounds, constraint handling is used to prevent the ES algorithm from exploring outside the bounds. The repair method used is defined as follows

while *value* x_i *violates bound* b_i **do**
| $x_i -= 1.1 * (x_i - b_i)$;
end

Algorithm 4: ES repair algorithm

It was also necessary to choose what type of PSO algorithm to implement; *gbest* or *lbest*. A *gbest* PSO was chosen, for simplicity. As with ES, the PSO algorithm also needed to be defined with a bound handling algorithm. The bound handling algorithm implemented is a modified version of velocity reflection, where the particle's position is reflected back into the search space by the amount it violated the bound (see algorithm 5, adapted from [46])

Once the specifics of the algorithms had been chosen, the structure of the library could be

while *value* x_i *violates bound* b_i **do**

  if b_i *is* x_{max} **then**

    $x_i = b_i - |x_i - b_i|$;

  else

    $x_i = b_i + |x_i - b_i|$;

  end

end

Algorithm 5: PSO bound handling algorithm

designed. From the research stage of the project, the general structure and flow of the algorithms were known (see fig. 2.3 and algorithm 2). This allowed the focus of the design process to be on the class structure of the system as opposed to the data flow or structure of the algorithms.

4.1.1 Class Structure

The full class diagram for the optimisation library can be seen in appendix A.1; many constructors and public/protected variables have been left out for compactness. The core of this design are the Population, Individual and Chromosome classes.

A Population consists of one or more Individuals, stored in a array defined with protected visibility. An array was chosen over one of the generic container classes, such as ArrayList, because the population size is always constant throughout and the system only requires read and write access, so the advantages of using a container class would not be exploited. The array was defined as protected in order to allow subclasses to access it simply and quickly. In order to meet various requirements of the system, the Population class provides functionality to save to and load from a file and store the results of optimisation and plot them on a scatter graph (using JFreeChart[47]).

An Individual contains one or more Chromosomes, stored in an array. Again, an array was chosen because the number of chromosomes is constant and only simple access is required. The class implements the Comparable interface because the population of individuals needs to be sortable; individuals are sorted by their normalised fitness value. The Individual class provides methods which allow an individual to write to and load from a file; this allows the Population class to defer this to the Individual class, making it easier for subclasses of Population to override the write and load methods as needed. The class also provides a copy constructor, which is

essential for recombination to prevent the parent individuals being modified by the operator.
The Chromosome class supports both real value and binary encoding. Both these encoding methods were included in one class, instead of a class hierarchy, for simplicity when constructing a population. The use of binary or real value coding is controlled by a boolean parameter passed to the Chromosome's constructor. As with the Individual class, the Chromosome has functionality which allows it to be written to and loaded from a file. The justification for this is that it allows the Individual to defer writing and loading to the Chromosome, so subclasses of Individual can easily override the functionality.
The design of these classes was chosen to allow population based algorithms to be added with minimal effort. This aim is also evident in the design of the genetic operator hierarchy. Using this design, a new population based algorithm can be added by following these steps:

- If the algorithm does not use real value or binary encoding, a subclass of Chromosome needs to be created which supports the encoding method required. A subclass of Individual also needs to be created to initialise the chromosome array with the new Chromosome subclass
- Implementations of the Recombination, Selection and Mutation need to be added as needed to provide the genetic operator functionality for optimisation
- A subclass of Population needs to be created, implementing the optimiseGeneration method, defining how to optimise each generation
- The write and read functionality in Chromosome, Individual and Population should be overridden as necessary

The Fitness class is used to provide fitness evaluation for a population. To add a fitness function, the class is simply extended and the evaluateFitness method implemented. This allows new fitness functions to be added easily. The fitness function to optimise is then passed into the Population constructor. Stopping criteria for a function can be added by subclassing Population and overriding the stopping method.

The configurable parameters of the algorithms, population size etc. are controlled by parameters in the constructors of the specific algorithm classes, i.e. GAPopulation. Since these have been excluded from the class diagram, they are listed here

```
GAPopulation(int populationSize, int numParents, int numChromosomes
    , int[] chromosomeLength, int numBits, double[] minBound, double[] maxBound
    , Fitness fitness, int opMode, int iterations, int selectionMode
    , int recombinationMode)
ESPopulation(int populationSize, int numParents, int numOffspring, int numChromosomes
    , int[] chromosomeLength, double[] minBound, double[] maxBound, Fitness fitness
```

```
    , int opMode, int iterations, int selectionMode
    , int recombinationMode)
PSOPopulation(int populationSize, int numChromosomes, int[] chromosomeLength
    , double[] minBound, double[] maxBound, Fitness fitness, int opMode, int iterations
    , double k)
```

4.1.2 Testing

Before the development of phase two could proceed, the results of phase one needed to be tested. Testing would be done using an arbitrary function chosen from a set of common functions[48]. The function chosen was De Jong's function 1, or sphere function. This was chosen because of its simplicity and the author's familiarity. This is defined as[48]

$$f_x = \sum_{i=1}^{n} x_i^2 \tag{4.1}$$

Next, the test parameters for the algorithms needed to be established. The bounds for the function are defined by De Jong as -5.12 and 5.12[48] and he also defines a suggested population size of 50 and 1000 as the maximum number of generations[23]. The chromosome length (number of genes) was defined as 20. This value was chosen because it is the default used in the Shark library examples. So the algorithms were bounded in the range[-5.12, 5.12] and use a population size of 50, a chromosome length of 20 and produce 1000 generations. The remaining parameters are algorithm specific.

4.1.2.1 Genetic Algorithm Parameters

De Jong suggests that the crossover rate should be 0.6 and the mutation rate 0.001[23]. The number of parents was 50, so that an offspring population of 50 was produced. The number of bits used to represent each gene was set at 10, because, as with the number of genes, this is the default used in the Shark library examples.
So the parameter setup of the genetic algorithm was:

- Population size = 50
- Number of generations = 1000
- Number of chromosomes = 1
- Number of genes = 20

- Number of bits to represent each gene = 10
- Crossover rate = 0.6
- Number of parents per generation = 50
- Mutation rate = 0.001

4.1.2.2 Evolution Strategies Parameters

The number of offspring to produce each generation was set to 50/100 for $(\mu + \lambda)$ and (μ, λ) selection respectively. The number of offspring needs to be higher for (μ, λ) selection to ensure diversity in the population. The number of parents used to produce the offspring was set to 2 for discrete and intermediate recombination modes. The design of the system allows any number of parents to be used in discrete and intermediate recombination modes; if the number of parents is greater than two then multi-recombination is performed. The global discrete and global intermediate modes use the entire population as parents. One strategy parameter was used and this was stored in a separate chromosome and initialised to 3 (default value in the Shark library examples).
So the parameter setup of the evolution strategies algorithm was:

- Population size = 50
- Number of generations = 1000
- Number of chromosomes = 2
- Number of genes = 20
- Number of strategy parameters = 1
- Number of parents per offspring = 2
- Number of offspring = 50 or100

4.1.2.3 Particle Swarm Optimisation Parameters

No examples for the value of the velocity clamping factor, k, could be found, so this was set by experiment. The PSO was run a number of times minimising De Jong's function with different

values of k, and a value of 0.25 was found to give a smooth and quick convergence. The algorithm also requires a separate chromosome of velocities, the same length as the position chromosome. The velocities where initialised to 0.
So the parameter setup of the particle swarm optimisation algorithm was:

- Population size = 50
- Number of generations = 1000
- Number of chromosomes = 2
- Number of genes = 20
- Number of velocities = 20
- Velocity clamping factor = 0.25

4.1.2.4 Test Process

All selection and recombination mode combinations would be tested for all algorithms, by maximising and minimising De Jong's function. Then, the best performing combination, with an easily available implementation in Shark, would be compared with a Shark implementation. The performance measure would be convergence speed; how quickly the population reached an optimum, and accuracy; both how close the population got to the true optimum and how close the system's optimum matched Shark's. Using the bounds -5.12 and 5.12 and 20 genes, this gives a maximum of 524.288 and a minimum of 0. These are the values used to measure accuracy.

4.2 Phase Two: The Arena

Once phase one was finished, the optimisation library could be adapted to play Connect 4 in The Arena.
The first stage in this process was to define how the population would be trained and the fitness function used to evaluate the population. Training would be carried out by playing the individuals in the population against each other. The fitness function was defined as

$$\text{maximise } f_x = \frac{actualWins(x)}{expectedWins(x)} - \frac{actualDraws(x)}{expectedDraws(x)} - \frac{actualLosses(x)}{expectedLosses(x)} \tag{4.2}$$

This function is designed to penalise a player who plays badly but was expected to play well worse than one who plays badly but was expected to play badly. The expected output of a game could be found using the state of the game after 8 moves. During the research for the project, a dataset was found which contained all possible 8 move combinations and the corresponding expected outcome[49]. Unfortunately, if a game finishes in less than 8 moves, then the expected output of the game cannot be found, and this will affect the fitness of the players involved.

Following this, the design and implementation of this phase was split into two: first, to get the optimisation library working in The Arena using simple random play and second to update the playing algorithm to be more intelligent and adjust the genotype of the individuals accordingly.

4.2.1 Initial Experiments

Before designing the extensions to the optimisation library, some initial experiments were carried out in order to understand The Arena better.
A number of player modules were produced, firstly by following the tutorials provided[50] and then independently. The tutorials introduce the basics of the classes involved and develop a module which plays randomly and outputs the state of the game board, the position and owner of each counter, each turn. Building on this information, a more intelligent player was produced, which crudely identified where there were two or three counters in a row and who they belonged to. It then attempted to build up its two/three in a rows whilst blocking the opponent's three in a rows. This module played significantly better than the random player and was capable of beating the author very rarely.

4.2.2 Class Structure

Appendix A.2 shows the additional classes that were added to provide The Arena functionality. The diagram is highly cut down for compactness; only the new classes and their super classes are shown. It can be assumed that any classes not shown are unchanged. The few additions that were needed to support this new problem and modify the algorithms proves that the phase one designs meets the non-functional requirements of the system; that new problems, algorithms and stopping criteria must be able to be added easily.

As can be seen, three new population classes were added, extending the algorithm populations, and these new classes implement the Arena interface Connect4Player. This means that each population class is a player module for The Arena. Each population has two pointers, one to the individual player1 is currently playing with and the other to the individual player2 is playing with. Player1's pointer is increased whenever player2 has played with all individuals, and when player1 has played with all individuals, the pointers are reset and the genetic operators applied to the population. This allows the population to play against itself in a tournament in The Arena, but means that $populationSize^2$ games are needed for each generation.
An alternative solution was considered, where each individual in the population was a player module and the population compiled the individuals into jar files containing a text file with its state. The population would then run a tournament with all the individuals, perform the necessary optimisation steps and then update the jar files with the new state of the individuals. This method was not chosen however, because of the author's unfamiliarity with manually compiling and updating jar files.

The GARecombination and ESRecombination classes were also extended, and these new classes override the copyIndividual method. This was necessary because the new GA and ES Population classes have populations of the new Connect4EAIndividual class. The original behaviour of copyIndividual returned a new instance of the Individual class, as a copy of the original. It was necessary to override this behaviour to return instances of Connect4EAIndividual to prevent ClassCastExceptions caused when the new Population classes attempt to cast the individuals in the population.

A new comparator was needed in order to allow the populations to be sorted by raw fitness value, instead of normalised fitness. This was needed because the Connect4Fitness class overrides the normalise function in order to deal with negative fitness values. Originally, the normalise function was not overridden. The original normalise function divided each individual's fitness by the sum of all fitnesses so all the fitness values added up to 1. The problem was that the negative fitness values possible with the fitness function defined in equation 4.2 would skew the normalised fitness values. To resolves this, the Connect4Fitness class overrides the normalise fitness function and uses a ranking method to normalise the fitness. The ranking method is identical to that described in section 2.3.5.2 and this requires that the population be sorted by

the original fitness values.

The DatasetParser class was added to parse the dataset mentioned before into a format that would allow the population classes to quickly search for a game state and find the expected outcome. This was originally implemented using a HashMap, mapping a GameState object to an int representing the expected outcome, 1 for a player1 win, -1 for a player2 win and 0 for a draw. A HashMap was chosen because it would give near O(1) lookup time complexity. However, when the parsing method was tested, it took over two minutes to parse the 67557 game states. Upon analysis, it found that creating the Map was taking all the time, and this was put down to the fact that the hashCode method had not been overridden for the GameState class in The Arena, leading to a large number of collisions when inserting into the map. To get around this, the dataset was parsed into an ArrayList instead of a HashMap. The ArrayList uses an inner class, Connect4ExpectedResult, to store the GameState and the expected result. This class overrides the equals method, to allow two Connect4ExpectedResults to be compared. By switching to an ArrayList, the author expected to see a large increase in lookup time, but when tested, parsing and lookup took between one and two seconds, a significant reduction on the two minutes it took using a HashMap.

4.2.2.1 Design Problems

The problem with the initial design for this phase was that it assumed that each player module was instantiated only once and then the playMove method was called every turn. This however is not the case. The Arena in fact instantiates the player modules every turn in order to prevent any information being shared between turns. This caused problems for the initial design, as it was originally parsing the Connect 4 dataset on instantiation, adding a 1-2 second overhead every turn, and also meant that the player pointers were reset to the first individual every turn. To get around the dataset problem, it was loaded as needed, i.e. after the 8^{th} move. This results in a 1-2 second overhead after the 8^{th} move only. To try and get around the pointer problem, the pointers, and other variables, were made static so they would be instantiated only once. This still did not resolve the problem, as The Arena resets static variables before instantiating the player module. The final workaround was to save and load the state of the population from a file. As such, the new Population classes had to override the write and load methods from Population. The new Individual classes also needed to override the reading and writing functionality from

their superclasses. The default behaviour of the Population classes was changed to instantiate the population with the chosen parameter setup (detailed later) and then the state of the population was loaded from the file. To minimise output, the file is only written a maximum of twice per game; after the 8^{th} move to update the individuals' expected outcomes and at the end of each game, when the individuals are updated with the actual outcome. As long as The Arena was run in insecure mode, this method worked and the population can be optimised.

4.2.3 Playing Algorithm

Unfortunately, due to the problems encountered whilst adapting the optimisation library to The Arena, there was little time left for phase 2 development before the start of phase 2 testing. As such, the author decided to use the player modules provided by The Arena as the basis of the playing algorithm. Both the playing performance of the different modules and the algorithm complexity were studied, and a simple, yet well performing algorithm was chosen.
The algorithm designed is based on that used by the *ChrisDefensePlayer2* module[50], and is an enhancement of the intelligent player that had been developed as part of the initial Arena experiments; it ranks each of the available moves, both offensively and defensively. The ranking is carried out using the following algorithm

```
for each move do
    best rank = 0;
    for i = 0 to 3 do
        rank the 4 counter line to the left of move.column - i;
        update best rank;
        rank the 4 counter line above move.row - i;
        update best rank;
        rank the 4 counter upper right diagonal from move.column - i, move.row - i;
        update best rank;
        rank the 4 counter lower right diagonal from move.column - i, move.row + i;
        update best rank;
    end
end
```

Algorithm 6: Move ranking algorithm

with each line ranked using the number of missing counters in the line. Because the column and row of the move is adjusted by 0-3, this allows us to only check for lines to the left, upper right and lower right, of the move. Consider the gamestate in figure 4.1. When ranking the line to the left of the move for the second column, the rank is two, since two counters are missing in that 4 counter line. However, when we adjust the column by subtracting one, the rank goes up to 3, as only one counter is now missing (in column two). When ranking offensively, lines of the

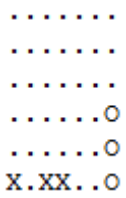


Figure 4.1: Sample gamestate

player's counters are considered and when ranking defensively, lines of the opponent's counters. Looking at figure 4.1 again, if it is player1's turn (represented by the X characters) then the highest ranked defensive move will be the move for column 7, since player2 (the O's) has three counters in that column.

Once each move is ranked, one of them is then selected and played. The selection is done according to various probabilities which make up the individual's genotype.

4.2.4 Designing the Genotype

Again, because of the time constraints caused by problems with adding The Arena functionality, the genotype of the individuals in the population was not as well developed as intended. The genotype chosen encoded the probabilities of selecting moves in a few simplistic situations. The author had hoped to try and encode some of the strategic rules defined by Victor Allis[1] in the genotype but was not able to because of time limitations.

The implemented genotype is made up of three chromosomes, the first two made up of four genes and the last made up of seven.

4.2.4.1 Chromosome One

The first chromosome encodes various probabilities for when the individual is player1.

- Gene 1: encodes the probability of playing a counter in the middle board position on the first move of the game. By playing in the middle position, it has been proven that player1 will always win the game if both players play perfectly from there on[1]
- Gene 2: encodes the probability of finishing a three in a row for player1
- Gene 3: encodes the probability of blocking a three in a row for player2
- Gene 4: encodes the probability of choosing a offensive move from the ranked move set

If it's the first move of the game, the probability in gene 1 is checked to see if player1 plays in the middle column, if not, then player1 plays a random move. Whenever there is a three in a row found, the corresponding probability is checked to see whether or not to block or finish it. Whenever no other move has been selected, the probability in gene 4 is checked, if it passes then an offensively ranked move is chosen, if not then a defensively ranked move is selected.

4.2.4.2 Chromosome Two

The second chromosome encodes various probabilities for when the individual is player1.

- Gene 1: encodes the probability of playing a counter in the middle board position on the first move of the game. This gene is never used, it is included only to make chromosome one and two the same length for easy coding
- Gene 2: encodes the probability of finishing a three in a row for player2
- Gene 3: encodes the probability of blocking a three in a row for player1
- Gene 3: encodes the probability of choosing a offensive move from the ranked move set

Whenever there is a three in a row found, the corresponding probability is checked to see whether or not to block or finish it. Whenever no other move has been selected, the probability in gene 4 is checked, if it passes then an offensively ranked move is chosen, if not then a defensively ranked move is selected.

4.2.4.3 Chromosome Three

The final chromosome simply encodes the probability of selecting each move in the ranked move set. Because there is no constraint that the values of the genes in this chromosome must add

up to exactly 1, the probabilities are scaled such that they add up to exactly 1 before they are used to select a move from the move set.

4.2.5 Testing

Once developed, the algorithms were tested with a small number of individuals (2-4) for 1000 generations to ensure that the algorithms were working as intended and optimising. It also gave some performance by which the training process could be planned. The real testing would come after the algorithms had been trained.

4.2.5.1 Training

The training process was the most important stage of the development of the project. It involved letting the algorithms optimise in The Arena. When the gantt chart for the project was drawn up, it was unknown how long the training process would take, so a period of four weeks was allocated. Now, with the performance benchmarks of the initial tests, an estimate of the training time could be calculated. The initial testing has shown that 400 games took approximately 20 minutes to run. If 400 games was taken to be one generation, this gave a population of 20 individuals. This gave an approximate training time for 1000 generations of two weeks.
Due to unforeseen circumstances, training started approximately a week late, so a population size of 20 was chosen since this would still leave a weeks contingency. It was not possible to train the algorithms in one large tournament, due to memory limitations, so instead it was trained in small tournaments, each covering 5-10% of the required games.
The execution of the first tournament with 20 individuals took longer than anticipated. The first tournament was of 20000 games, 5% of the total number of games and it took over a day to finish. Extrapolating that execution time over the rest of the training process, It looked like the training process would overrun. As such, the population size was reduced to 10 individuals. This allowed the training process to finish on time, but may have limited the optimisation of the algorithms.

4.2.5.2 Training Parameters

As mentioned above, the population size was set to 10 individuals, the number of chromosomes to 3, with 4, 4 and 7 genes respectively. The remaining parameters were kept similar to those

used in testing phase one (see section 4.1.2). The selection and recombination modes used to compare the project to Shark implementations were chosen for phase 2 testing. The complete list of testing parameters will be listed in the next section.

4.2.5.3 Testing Process

After the training period was completed, the system could be be tested to see how effectively it had learnt to play Connect 4. This would be done in two ways, firstly each algorithm would play 50 games against the author; 25 as player1, 25 as player2, and secondly, each algorithm would play 1000 games against the recursive player module provided by The Arena; 500 as player1, 500 as player2. This module is known to play very well, especially when given a large turn limit. Various turn lengths would be used to limit the performance of the recursive player and to try and find a time limit where the algorithm matched the recursive player. For each algorithm, only the best individual was used when playing against the author and the recursive module.

Chapter 5

Evaluation

Sections 4.1.2 and 4.2.5 detail the testing process for the two development phases of the project. This section will present the results of testing and evaluate the success of the project using these results.

5.1 Phase One

5.1.1 Genetic Algorithm

5.1.1.1 Test Setup

The parameters used to test the genetic algorithm are listed below. These are discussed in section 4.1.2.1.

- Fitness function = De Jong's function 1
- x_{min} = -5.12
- x_{max} = 5.12
- Population size = 50
- Number of generations = 1000
- Number of chromosomes = 1
- Number of genes = 20

- Number of bits to represent each gene = 10
- Number of parents = 50
- Crossover rate = 0.6
- Mutation rate = 0.001

5.1.1.2 Binary Tournament Selection Results

The optimisation graphs for optimising De Jong's function 1 with binary tournament selection are shown in appendix B.1.1 and B.1.2. Looking at the graphs, it can be seen that the genetic algorithm has performed well at optimising De Jong's function 1. From the graphs, the GA locates the area containing the optimum within 200-300 generations. For maximisation, that optimum looks to be close to 525, and for minimisation, close to 0. Considering that the maximum value for De Jong's function in the range [-5.12, 5.12] is 524.288 and the minimum 0, the genetic algorithm has converged quickly to a good optimum. The output of the genetic algorithm supports the data shown by the graph. Each iteration, the GA outputs the best value in its current population (this output has not been included for compactness). The output of the GA shows that it optimised to an optimum of 524.288 for maximisation and 5.00977×10^{-4} for minimisation. The speed at which these optima were reached differed between one and two-point crossover. For one-point crossover, the minimum was reached in 385 generations and the maximum in 328. For two-point crossover, the minimum was reached in 318 and the maximum 321. Two-point crossover converged slightly faster, which can be explained by the fact that, in general, two-point is going to introduce less diversity into the population, as fewer bits will be exchanged between parent individuals. This will reduce how explorative the population is.

5.1.1.3 Rank Selection Results

The optimisation graphs for optimising De Jong's function 1 with rank selection are shown in appendix B.1.3 and B.1.4. As with binary tournament selection the graphs show that the genetic algorithm has optimised De Jong's function 1 well; locating the are containing the optimum within 200-300 generations. For maximisation, that optimum looks to be close to 525, and for minimisation, close to 0, so the genetic algorithm has converged quickly to a good optimum. The output of the genetic algorithm supports the data shown by the graph; a maximum of 524.288

was reached and a minimum of 5.00977×10^{-4} reached, these are the same maximum and minimum binary tournament found. As with binary tournament selection, the choice of one or two-point crossover affects the convergence speed of the algorithm. For one-point crossover, the minimum was reached in 314 generations and the maximum in 210 and for two-point crossover, the minimum was reached in 286 and the maximum 309.

5.1.1.4 Proportional Selection Results

The results of the proportional selection tests can be seen in appendix B.1.5 and B.1.6. In comparison with the graphs for binary tournament and rank selection, the proportional graphs show a poorly performing algorithm. For maximisation, there is wide diversity in the population and a maximum of less than 500 has been reached. For minimisation, the graph shows the population has been maximised, which suggests the scaling method is not working effectively. After a number of experiments, it was found that the proportional selection GA implemented is greatly affected by the number of parents parameter, and therefore the number of offspring produced by crossover. By adjusting the value of this parameter, the performance of the GA can be greatly improved, as shown by appendix B.1.7. For minimisation the GA is now using 10 parents and for maximisation 20. The performance of the GA now matches the performance of binary tournament and rank selection. A maximum of 524.288 has been found and a minimum of 3.30956. Whilst the accuracy has been improved, the convergence speed is still significantly worse than that of the other selection methods; maximisation took 706 generations and minimisation 881.

5.1.1.5 Comparison With Shark

For the next stage of testing, the comparison between the author's implementation and a similar implementation in the Shark machine learning library, a genetic algorithm with binary tournament selection and two-point crossover was chosen. Whilst rank selection had been shown to have slightly better performance to binary tournament selection, the author was unable to find a similar implementation of the linear ranking method in the Shark library, so binary tournament selection was chosen to ensure a fair comparison. Two-point crossover was chosen as it had resulted in slightly better performance over one-point crossover with binary tournament selection. The results of optimisation with the Shark implementation can be found in appendix B.1.8.

Comparing this to the graphs in appendix B.1.1, the Shark implementation seems to perform similarly to the project's implementation. Looking at the actual output of the Shark implementation, the algorithm has found a minimum of 5.00977×10^{-4} and a maximum of 524.288, which are the same as those found by the project's implementation. The convergence speeds are also similar, with the Shark implementation taking 310 and 324 generations for minimisation and maximisation.

5.1.1.6 Evaluation

The genetic algorithm has been implemented successfully to a strong degree. In accordance with the requirements of the system (section 3.1), the genetic algorithm has been implemented supporting a number of selection and recombination modes, and the modes to be used, as well as the other configurable parameters, can be easily changed through the algorithm's constructor. The majority of the supported selection modes perform very well, with similar accuracy and convergence speed as a similar implementation in the Shark machine learning library. The proportional selection mode is the exception, as it performs very badly. However, through careful parameter tuning, proportional selection can be made to perform significantly better, but still worse than the other selection modes. The algorithm is able to optimise De Jong's function 1 very well, actually reaching the maximum value and finding a very small minimum: 5.00977×10^{-4}. The algorithm is unable to minimise any lower than 5.00977×10^{-4} because at that point, the value of the genes are as small as they can be given the number of bits used to represent them. Because of how the decoding equation works, the genes are only able to take certain values, determined by the bounds of the function and the number of bits that represent each gene. By increasing the number of bits, the precision of the genes would be increased, and therefore smaller minima could be found.

5.1.2 Evolution Strategies

5.1.2.1 Test Setup

The test setup for the evolution strategies algorithm is discussed in section 4.1.2.2, it is listed again here for ease of reference.

- Fitness function = De Jong's function 1

- x_{min} = -5.12
- x_{max} = 5.12
- Population size = 50
- Number of generations = 1000
- Number of chromosomes = 2
- Number of genes = 20
- Number of strategy parameters = 1
- Number of parents per offspring = 2
- Number of offspring = 50 or100

5.1.2.2 (μ, λ) Selection Results

The graphs in appendix B.2.1, B.2.2, B.2.3 and B.2.4 show the results of optimising De Jong's function 1 using a (μ, λ) evolution strategy algorithm. Looking at the graphs it is clear that the convergence speed of (μ, λ) ES is better than that of GAs; the area containing the optimum is reached within 100-200 generations. It is also clear that the performance of intermediate recombination is completely undefined for maximisation problems in this implementation. Looking at the output of the algorithm, when minimising very small minima are found. For discrete and global discrete recombination, minima of 3.7767×10^{-21} and 5.7056×10^{-14} were found and for intermediate and global intermediate, minima of 3.6096×10^{-44} and 9.2004×10^{-41}. After the 1000 generations, the minimum was still decreasing so if more generations were produced the minima found by the algorithm would get even smaller, and may eventually reach the true minimum of 0. For maximisation, the maximum value of 524.288 was reached by both discrete and global discrete recombination, with a convergence speed of 289 and 233 generations respectively.

5.1.2.3 $(\mu + \lambda)$ Selection Results

Appendix B.2.5, B.2.6, B.2.7 and B.2.8 show the results of optimising using the $(\mu + \lambda)$ evolution strategy algorithm. The performance of $(\mu + \lambda)$ is very similar to that of (μ, λ). The area

containing the optimum is located within 100-200 generations and intermediate and global intermediate recombination still performs badly for maximisation; the diversity is lesser, but the true maximum is still not found. Again, very small minima are found; 6.535×10^{-41}, 1.2289×10^{-38}, 1.4018×10^{-43} and 1.8098×10^{-42} for discrete, global discrete, intermediate and global intermediate recombination, and as with (μ, λ), if the number of generations were increased, smaller minima would be found. For maximisation, discrete and global discrete recombination both reach 524.288, requiring 345 and 289 generations to do so.

5.1.2.4 Comparison with Shark

For the comparison with the Shark implementation, a $(\mu + \lambda)$ ES with discrete recombination was chosen. $(\mu + \lambda)$ selection was chosen, because whilst its convergence speed was slightly slower that (μ, λ), it requires the generation of less offspring which makes the production of each generation more efficient. Discrete recombination was chosen because it was one of the two recombination modes that worked for both minimisation and maximisation. It was chosen over global discrete recombination because the Shark library does not provide a method to perform global recombination by default. The results of optimisation using the Shark implementation can be seen in appendix B.2.9. The performance of the Shark implementation and the project's are vary similar. For minimisation, both identify the area containing the optimum within 100-200 generations and reach very low minima, the Shark implementation reaching 3.91939×10^{-17}. Both implementations would also find smaller minima if the number of generations was increased. For maximisation, the same is true, the area containing the maximum is found with 100-200 generations and then the maximum of 524.288 reached within another 100 generations or so, the Shark implementation taking 309 generations in total.

5.1.2.5 Evaluation

The evolution strategies algorithm has been implemented effectively but with some problems. As per the project requirements (section 3.1), a number of different selection and recombination modes are supported and these, along with the other configurable parameters, can be easily changed through the algorithm's constructor. The algorithm is able to optimise De Jong's function 1 extremely well, finding the maximum of 524.288 and finding very small minima. This shows that another system requirement has been met; the algorithm has not explored outside the

bounds of the function, showing that the constraint handling mechanism works successfully. It is still able to find values on the bounds of the search space too, again showing that the constraint handling method is effective. Unlike the genetic algorithm, where the accuracy of the minimum is limited by the number of bits used to represent each gene, because ES use real value coding there is no limit on how small the values of the genes can become; the only limiting factor to the minimum identified by the algorithm is the number of generations. Not only is the accuracy of the ES algorithm good, but the convergence speed is fast too, typically identifying the area containing the optimum in 100-200 generations and then identifying the optimum in less than 350. The implementation of ES also matches the performance of a Shark implementation very closely. The limitation with this ES implementation is that the performance of intermediate and global intermediate recombination is undefined for maximisation problems.

5.1.3 Particle Swarm Optimisation

5.1.3.1 Test Setup

The test setup for particle swarm optimisation is discussed in section 4.1.2.3, it is listed again here for ease of reference.

- Fitness function = De Jong's function 1
- x_{min} = -5.12
- x_{max} = 5.12
- Population size = 50
- Number of generations = 1000
- Number of chromosomes = 2
- Number of genes = 20
- Number of velocities = 20
- Velocity clamping factor = 0.25

5.1.3.2 Optimisation Results

Appendix B.3.1 shows the optimisation graphs for the particle swarm optimisation algorithm. The graphs show a wide diversity in the population during the first 100-200 generations before the diversity steadily decreases as the population moves towards the optimum. This sort of pattern is expected, as the particles are initially distributed evenly in the search space and slowly move towards the best value found so far. When maximising De Jong's function 1, the population split into two branches; one converging on the optimum and the other remaining between 75 and 200. This suggests that PSO may have problems solving more complex maximisation problems. The accuracy of the algorithm is good, reaching the maximum of 524.288 and a minimum of 3.8202×10^{-10}. As with evolution strategies, the accuracy of the minimum value is limited by the number of generations; if the number of generations were increased then smaller minima could be found. The graphs clearly show that the convergence speed of PSO is much slower than that of genetic algorithms or evolution strategies, with maximisation taking 929 generations to reach the maximum. This is to be expected however, because of the way in which the algorithm works.

5.1.3.3 Comparison With Shark

Appendix B.3.2 shows the optimisation results of a PSO implemented in Shark. The performance of the Shark implementation is almost identical to the implementation in this project. A similar pattern if produced in the graphs, with the population starting off diverse and then steadily converging, and the population splitting into two branches when solving a maximisation problem. The accuracy of the two implementations are similar, both reaching the maximum 524.288, and the Shark implementation reaching a minimum of 3.89969×10^{-8}, again limited by the number of generations. The convergence speed is near identical as well, with the Shark implementation taking ever so slightly less generations to reach the maximum, 921 as opposed to 929.

5.1.3.4 Evaluation

The implementation of particle swarm optimisation is highly effective. Both maximisation and minimisation problems are solved with the same accuracy as that of the genetic and evolution strategies algorithms and the implementation matches very closely the equivalent in Shark. Again, minimisation accuracy is limited only by the number of generations, giving PSO an

advantage over GAs for minimisation. The convergence speed is significantly slower when compared with GAs or ES, but this is to be expected given the operation of the PSO algorithm. The only limitation is that when maximising the population split into two branches, one of which does not converge. This suggest that the algorithm could have problems optimising more complex maximisation problems.

5.1.4 Evaluation of Phase One

One of the objectives of this project was to "Implement and test the chosen population based optimisation algorithms, producing a generic implementation". The algorithms chosen were the genetic, evolutions strategies and particle swarm optimisation algorithms and the results of testing these algorithms with De Jong's function 1 are presented and discussed above. The results show that the three algorithms have been successfully implemented and perform well at optimising arbitrary maximisation or minimisation problems. In accordance with the requirements of the project, a number of selection and recombination modes are supported, the majority of which perform extremely well. Some of these modes do not work as expected; proportional selection and intermediate recombination, and this limits the success of the optimisation library slightly. The optimisation library developed in phase 1 also meets many of the other project requirements; the algorithms optimise only within the bounds of the problem, the optimisation results can be plotted to a scatter graph once optimisation is finished, the algorithms can be easily configured through their constructors and new problems, algorithms and stopping criteria can be easily added by overriding a few methods in the class hierarchy. In general, the system produced in phase one meets the project objective and requirements to a strong extent, limited only by the fact that a small number of the selection and recombination modes do not work as intended.

Another objective of the project was to "Compare the effectiveness and efficiency of the different algorithms". The test results obtained show a number of differences between the different algorithms.

With regards to efficiency, or convergence speed, the results show that evolution strategies is the fastest algorithm, generally finding the area containing the optimum in 100-200 generations and finding the optimum in less than 350. The genetic algorithm is the next fastest, taking only slightly longer than evolution strategies. The slowest algorithm by far is particle swarm

optimisation, taking over 900 generations to find the optimum. This is to be expected though, considering the operation of the particle swarm optimisation algorithm.

Regarding effectiveness, or accuracy, for maximisation the results obtained show that all the algorithms have the same accuracy, all of them identifying the true maximum value of 524.288. For minimisation however, the genetic algorithm was shown to have the lowest accuracy, only finding a value of 5.00977×10^{-4}. This however is the smallest value the algorithm could find given a 10 bit representation per gene; if the number of bits per gene were increased then smaller minima could be found. As for the accuracy of evolution strategies and particle swarm optimisation, they were more accurate than the genetic algorithm, and their accuracy was limited by the number of generation produced. When 1000 generations were reached, both of the algorithms were still finding smaller and smaller minima; if the number of generations were increased, even smaller values would be found, and the true minimum of 0 may even be reached.

5.2 Phase Two

5.2.1 Genetic Algorithm

5.2.1.1 Test Setup

The parameters used for testing the genetic algorithm are listed below

- Fitness function = function 4.2
- $x_{min} = 0$
- $x_{max} = 1$
- Population size = 10
- Number of generations = 1000
- Number of chromosomes = 3
- Number of genes = 4, 4, 7
- Number of bits to represent each gene = 10
- Selection mode = binary tournament

	Genetic Algorithm Wins	Author Wins	Draws
GA player1	6	18	1
GA player2	0	25	0

Table 5.1: Genetic algorithm Connect 4 results against the author

Time Limit	Genetic Algorithm Wins	Recursive Wins	Draws
5 seconds	3	994	3
1 second	12	972	16
0.5 seconds	10	963	27
0.25 seconds	13	968	19
0.15 seconds	35	930	35

Table 5.2: Genetic algorithm Connect 4 results against the recursive module

- Number of parents = 10
- Crossover mode = two-point
- Crossover rate = 0.6
- Number of parents per generation = 10
- Mutation rate = 0.001

5.2.1.2 Testing Results

Table 5.1 shows the results of the 50 games the genetic algorithm played against the author. It is clear from these results that the genetic algorithm is not playing Connect 4 especially well. It plays slightly better when playing as player1, but was still not capable of beating the author. The majority of the games were very short, typically finishing within 20 moves. Whilst playing the algorithm, the author noticed several problems with the way it was playing. The first, and most critical, was that the genetic algorithm played in patterns; that is, if the author played counters in certain places then the algorithm would always play the same counters in the same order. This made the algorithm very exploitable and therefore, it was extremely easy for the author to win. This pattern was mostly noticeable when the genetic algorithm was player2. A second defect the author noticed was that the algorithm would not always complete its three in a rows, or block the author's. This shows that the algorithm has not learnt the game effectively, as the probability to finish a three in a row, whether the opponents or yours, should be 1.

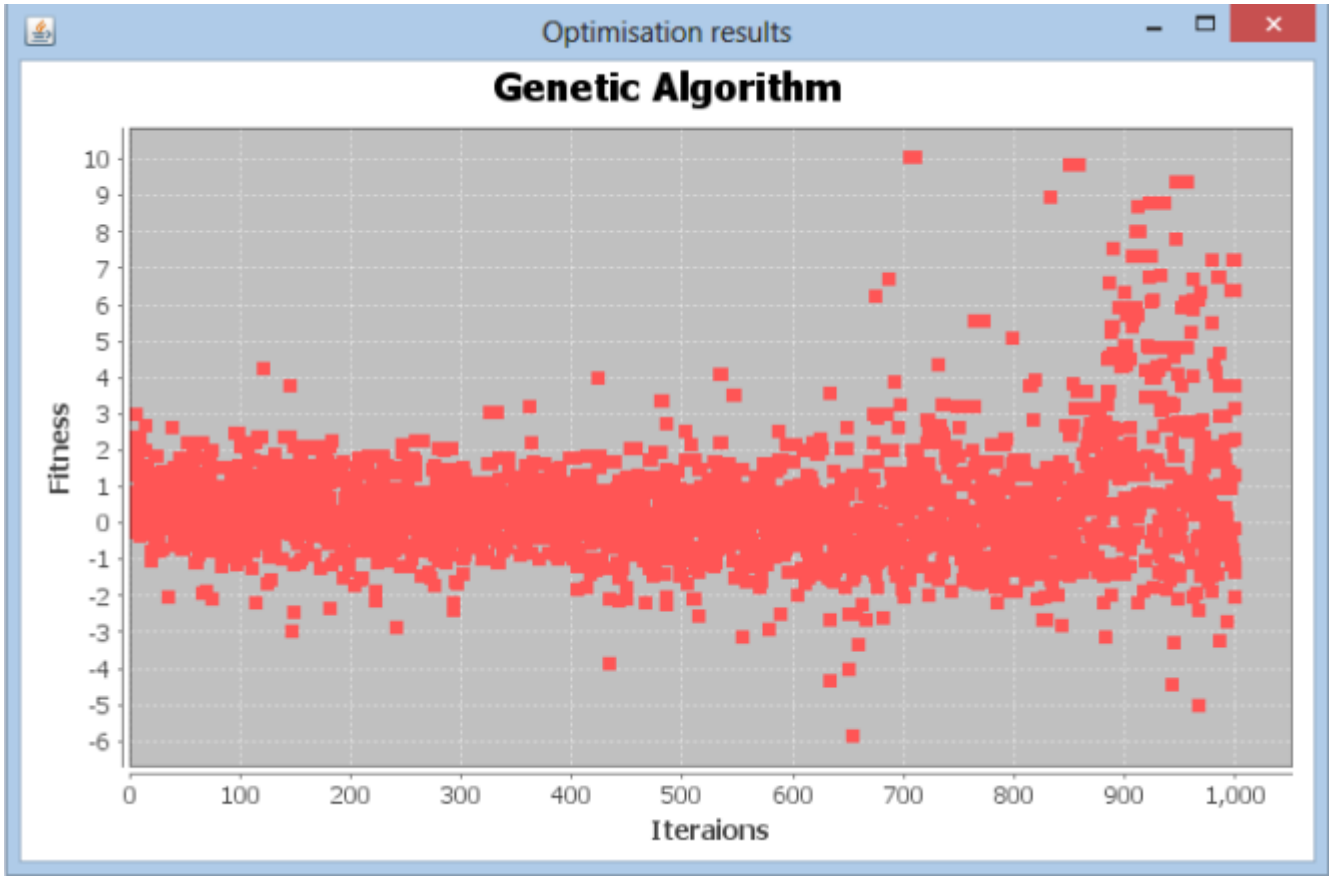


Figure 5.1: Genetic algorithm optimisation results for Connect 4

The results of the 1000 games against the recursive module (table 5.2) also show a poor playing performance by the genetic algorithm. The playing performance increases as the turn time decreases, which was expected, but the algorithm was not able to evenly closely match the performance of the recursive module. At a time limit of 0.15 seconds, the algorithm's performance was significantly higher than the others which suggests that at even smaller time limits the algorithm may be able to match, or even out perform, the recursive module. Unfortunately, it is not possible to test smaller time limits, as below 0.15 seconds the genetic algorithm exceeds the time limit and the recursive player wins by default.

Looking at the optimisation results of the genetic algorithm, shown in figure 5.1, it is clear that the population has not been optimised at all, which explains why the playing performance is poor.

5.2.2 Evolution Strategies

5.2.2.1 Test Setup

The parameters used for testing evolution strategies are listed below

- Fitness function = function 4.2
- $x_{min} = 0$
- $x_{max} = 1$
- Population size = 10
- Number of generations = 1000
- Number of chromosomes = 4
- Number of genes = 4, 4, 7
- Number of strategy parameters = 1
- Selection mode = $(\mu + \lambda)$
- Recombination mode = Discrete recombination
- Number of parents per offspring = 5
- Number of offspring = 10

5.2.2.2 Testing Results

Table 5.3 shows the results of the 50 games played by the evolution strategies algorithm against the author. From these results, it can be seen that the evolution strategies algorithm has learnt to play Connect 4 better than the genetic algorithm. The algorithm plays best when it is player1, actually beating the author over the 25 games. Whilst playing the algorithm, the author noticed several things. Firstly, that the games were significantly longer than those against the genetic algorithm, often nearly filling the entire board. Secondly, that the algorithm was capable of setting traps to guarantee that it won; i.e. playing so that the author had to play two counters in the same row to prevent it from winning or forcing the author to block in one column which enabled the algorithm to win next turn. Both of these features suggest that the algorithm

	Evolution Strategy Wins	Author Wins	Draws
ES player1	15	10	0
ES player2	6	19	0

Table 5.3: Evolution strategies Connect 4 results against the author

Time Limit	Evolution Strategy Wins	Recursive Wins	Draws
5 seconds	0	999	1
1 second	3	993	4
0.5 seconds	11	987	2
0.25 seconds	28	966	6
0.15 seconds	40	947	13

Table 5.4: Evolution strategies Connect 4 results against the recursive module

has learnt to play Connect4 to a good extent. There were however still some problems the author noticed; the algorithm still occasionally played in patterns which could be exploited and occasionally didn't block the author's three in a rows. Both of these limitations show that the algorithm has not completely learnt to play Connect 4.

The results of the 1000 games against the recursive module (table 5.4) show a slightly different story. The playing performance still increases as the turn time decreases, as expected, but overall the performance is worse than that of the genetic algorithm. ES wins more games than the genetic algorithm, but overall the recursive module wins more when playing against the ES algorithm. The optimisation results, shown in figure 5.2, could help explain this, as they show that some maximisation has taken place in the population. This means that the algorithm has learnt the playing algorithm to an extent and therefore will play less randomly. This lack of randomness may be the reason why evolution strategies performs worse against the recursive module than the genetic algorithm, as the random aspect of the GA may be harder for the recursive module to predict. As with the genetic algorithm results, performance significantly increases at 0.15 seconds but it is not possible to test lower time limits, since the algorithm will exceed the time limit.

5.2.3 Particle Swarm Optimisation

5.2.3.1 Test Setup

The parameters used for testing particle swarm optimisation are listed below

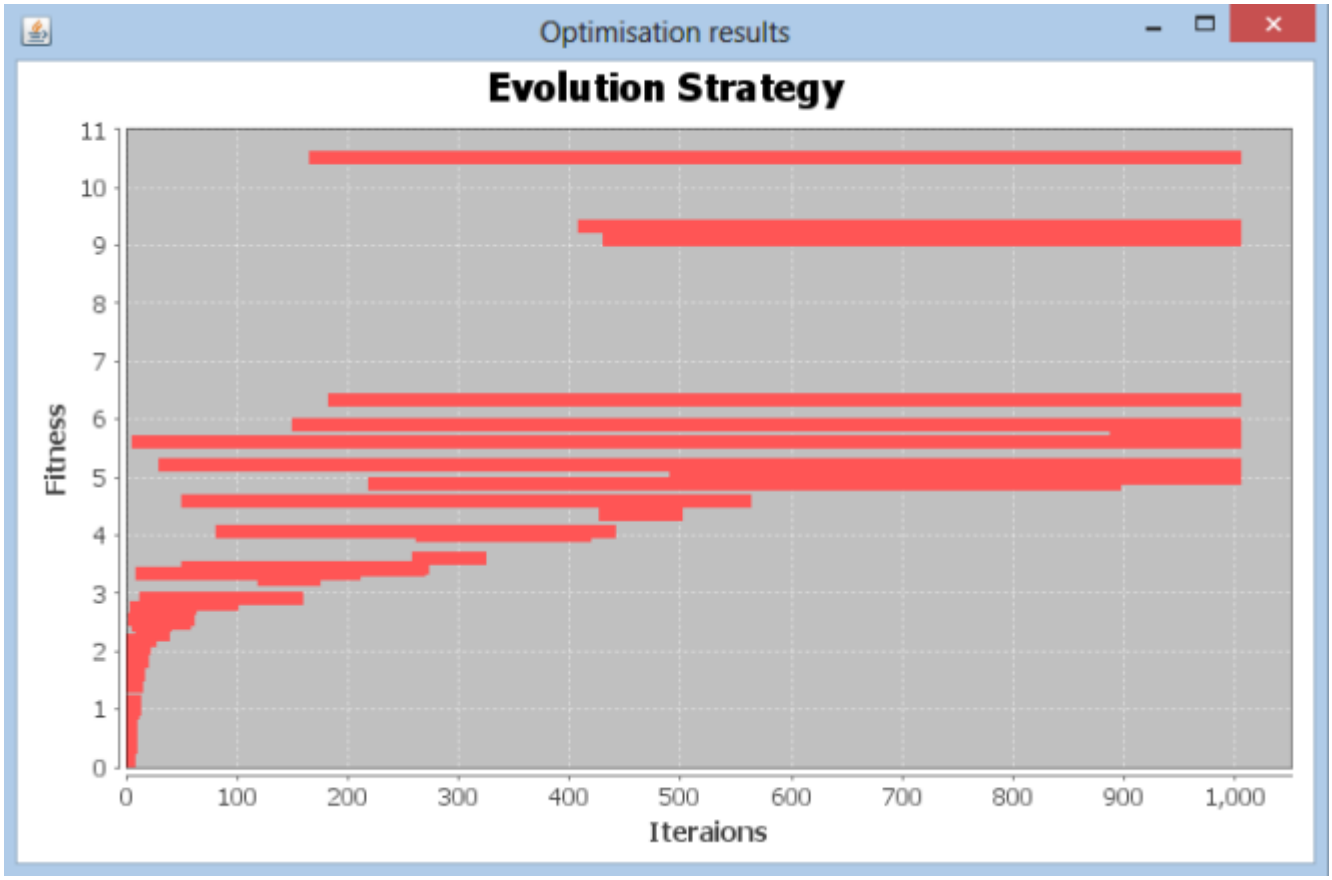


Figure 5.2: Evolution strategies optimisation results for Connect 4

- Fitness function = function 4.2
- $x_{min} = 0$
- $x_{max} = 1$
- Population size = 10
- Number of generations = 1000
- Number of chromosomes = 4
- Number of genes = 4, 4, 7
- Number of velocities = 7
- Velocity clamping factor = 0.25

	Particle Swarm Optimisation Wins	Author Wins	Draws
PSO player1	7	18	0
PSO player2	7	18	0

Table 5.5: Particle swarm optimisation Connect 4 results against the author

Time Limit	Particle Swarm Optimisation Wins	Recursive Wins	Draws
5 seconds	1	995	4
1 second	2	997	1
0.5 seconds	11	985	4
0.25 seconds	10	986	4
0.15 seconds	17	980	3

Table 5.6: Particle swarm optimisation Connect 4 results against the recursive module

5.2.3.2 Testing Results

The results of the 50 games particle swarm optimisation played against the author are shown in table 5.5. From these results, it can be seen that particle swarm optimisation plays slightly better than the genetic algorithm, able to win games against the author when player2, but not as well as evolution strategies. As with evolution strategies, the games against this algorithm were long, and it exhibited some ability to set traps, both of which suggest a good level of learning. However, the algorithm still did not always block the author so it cannot have learnt to play Connect 4 fully.

The results of the 1000 games against the recursive module (table 5.6) show a poor performance too. As with the other two algorithms, performance increases as the turn time decreases but the performance of particle swarm optimisation is the worse of any of the three algorithms. The optimisation results of particle swarm optimisation, figure 5.3, are similar to that of the genetic algorithm, except clustered around an even smaller area. This not only shows that the population has not been optimised but also that there has been only weak exploration, which could explain why the algorithm performed worse than the genetic algorithm, as there will be less randomness in the population.

5.2.4 Evaluation of Phase Two

The two project objectives the second phase of this project aimed to achieve were "Adapt the generic implementation to work with Connect 4 in The Arena" and "Evaluate the effectiveness

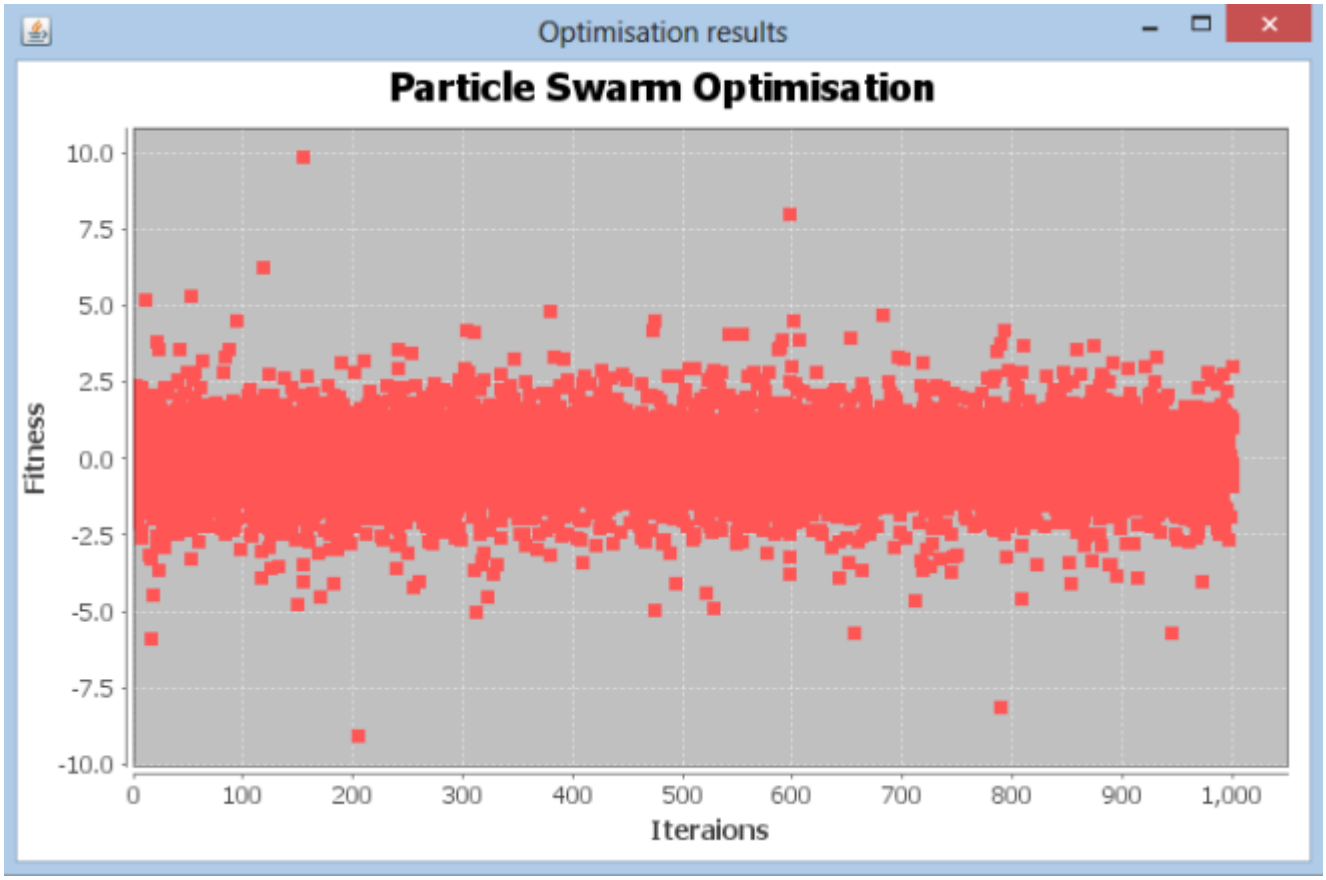


Figure 5.3: Particle swarm optimisation optimisation results for Connect 4

of the different algorithms in playing Connect 4". With regards to the first objective, this has been achieved strongly, as all three algorithms are capable of playing Connect 4 in The Arena and performing optimisation using Connect 4 as a fitness function. The effectiveness of this optimisation and how the algorithms play Connect 4 is questionable however.

When playing against the author, evolution strategies is the strongest algorithm, especially when player1, actually beating the author. The genetic algorithm is clearly the worst algorithm, incapable of winning any games when player2 and only winning 6 when player1, and particle swarm optimisation performs only slightly better. The optimisation graphs for the algorithms support this as they show that evolution strategies was the only algorithm to undergo any form on optimisation, even if that optimisation is extremely limited. The graphs for particle swarm optimisation and the genetic algorithm are very similar, both showing a wide diversity and no optimisation, hence why they performed similarly against the author.

The results of the tests to match the playing performance of the recursive player show that none of the algorithms were able to even closely match the performance of the recursive player. Again,

evolution strategies won the most games but overall the genetic algorithm plays best against the recursive player; with more wins and draws more than any other algorithm. The particle swarm optimisation algorithm is the worst performing. Looking at the optimisation graphs, the results suggest that the randomness the genetic algorithm has due to poor optimisation works better against the recursive player than the rules evolution strategies has learnt. The performance of each algorithm increased as the turn limit was decreased, which was expected as it limits the effectiveness of the recursive player, and increased significantly at 0.15 seconds for the evolution strategies and genetic algorithms. This suggests that at even lower turn limits, these algorithms may be able to match, or even exceed, the recursive player. Unfortunately it is not possible to test any smaller time limits, as the algorithms start to exceed smaller time limits than 0.15 seconds and the recursive player wins by default.

Chapter 6

Conclusion

This project aimed to investigate how well stochastic optimisation techniques could learn to play Connect 4. Specifically, three population based computational intelligence techniques were investigated; the genetic, evolution strategies and particle swarm optimisation algorithms.

After the initial research, the development of the project was split into two phases; firstly to implement the chosen algorithms into a generic optimisation library and secondly to adapt this library to play Connect 4 in The Arena. This allows the objectives of the project to be evaluated separately.

The first developmental stage aimed to achieve the following project objective: "Implement and test the chosen population based optimisation algorithms, producing a generic implementation". The extent to which this objective was met was evaluated by testing the performance of each algorithm in maximising and minimising De Jong's function 1. The test results for this phase show that this objective has been strongly achieved; efficient and accurate implementations of the chosen algorithms have been produced, able to maximise and minimise with a number of selection and recombination modes and the design of the system allows new functions and algorithms to be added with minimal effort. The only limitation to the success of phase 1 is the fact that a small number of the selection and recombination modes have either undefined or poor performance.

The objectives of the second phase of development were to "Adapt the generic implementation to work with Connect 4 in The Arena" and then to "Evaluate the effectiveness of the different algorithms in playing Connect 4". The evaluation of the algorithms' effectiveness would be tested by playing a number of games against both the author and the recursive player module provided

by The Arena. The adaptation to The Arena was achieved successfully, with the algorithms able to play Connect 4 and perform the necessary optimisation steps using Connect 4 as a fitness function. The adaptation of the algorithms to The Arena took longer that anticipated however, and therefore the development of an effective playing algorithm was limited, because of the limited time frame this project had for development. As such, the effectiveness of the project was expected to be hindered.

The test results show that only one of the algorithms, evolution strategies, was able to effectively learn to play Connect 4, when playing against the author, and even then only when playing as player1. None of the algorithms were able to closely match the recursive player, even at low time limits. This may show that the algorithms are not appropriate for playing Connect 4, or it may be that the design of the playing algorithm or the configuration parameters for the algorithms were limiting factors. More rigorous testing would be needed to produce more conclusive results.

Throughout the testing of the two developmental phases, the performance of the algorithms have been compared to meet the final objective of the project: "Compare the effectiveness and efficiency of the different algorithms". During phase 1 testing, the accuracy (effectiveness) and convergence speed (efficiency) of the algorithms were compared. It was found that evolution strategies was the most effective and efficient algorithm; it was able to find the maximum within 350 generations and its accuracy was limited only by the number of generations. Particle swarm optimisation was the least efficient algorithm, taken 900 generations to find the maximum, but considering the operation of the algorithm, this was expected. It was however, the second most accurate algorithm, again limited only by the number of generations. The genetic algorithm was shown to be the least accurate, only able to able find a minimum of 5.00997×10^{-4}, however this minimum was limited by the number of bits used to represent each gene, so if more bits were used smaller minima could be found. It was also the second most efficient algorithm, taking only slightly longer than evolution strategies to optimise the function. During phase two testing, the effectiveness of each algorithm at playing Connect 4 was compared, and again evolution strategies was shown to be the most effective, actually capable of beating the author and showing some clever behaviour in setting traps to ensure it won.

Overall, this project has achieved mixed results. On the one hand, the objective of developing a generic optimisation library using population based techniques has been almost wholeheartedly met and this library has been adapted to allow optimisation of rules for playing Connect 4. The results of this optimisation though have been poor, with the algorithms mostly unable to play

Connect 4 with any real effectiveness. Whether this poor performance is because of the time constraints of the project limiting the training period and design of the playing algorithm, or that the system design or algorithms themselves are inappropriate for Connect 4 is unknown and would need more rigorous testing to establish. If the focus of the project had been more towards the Connect 4 side of the problem, and not the implementation of the algorithms themselves, better results may have been achieved.

6.1 Future Work

Because of the mixed results obtained by the project, there is scope for future research.

Firstly, the system as it currently is should be more rigorously tested, with longer training periods and different parameter setups. This would allow the design of the system to be more effectively evaluated and establish whether it needs to be redesigned.

Now the algorithms have been adapted to optimise in The Arena, more time should be spent designing an effective playing algorithm and genotype for the individuals. This should be rigorously tested, and then a more conclusive answers as to whether the chosen algorithms are appropriate for playing Connect 4 could be obtained. The optimisation library could be adapted as well, to include more selection and recombination modes, which could improve the optimisation performance of the algorithms and they may play better as a result.

Other computational intelligence techniques could be developed to play Connect 4, in order to show whether computational intelligence is something that can be effectively applied to Connect 4 at all.

6.2 Closing Remarks

This project, whilst achieving slightly disappointing results, has been an interesting undertaking. Perhaps, if a computational intelligence library had been used, instead of implementing the algorithms manually, then better results could have been achieved, as more time could have been spent design the playing algorithm and genotype. With hindsight, perhaps this path

should have been taken but one of the author's main aims when suggesting this project was to gain practical experience implementing computational intelligence techniques, and in that respect this project has been very successful.

Appendix A

Class Diagrams

A.1 Optimisation library Class Diagram

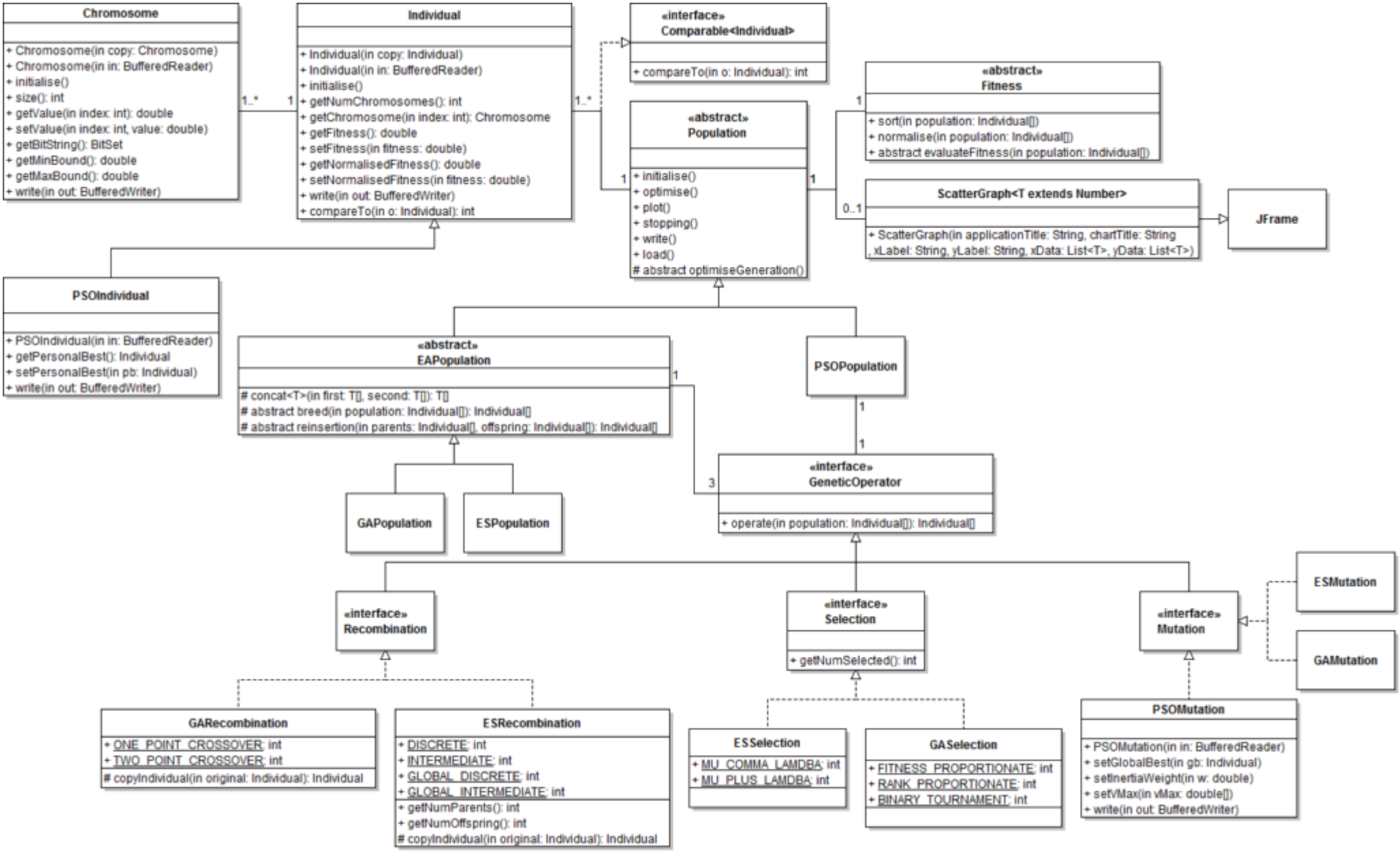


Figure A.1: Class diagram for the optimisation library

A.2 The Arena Class Diagram

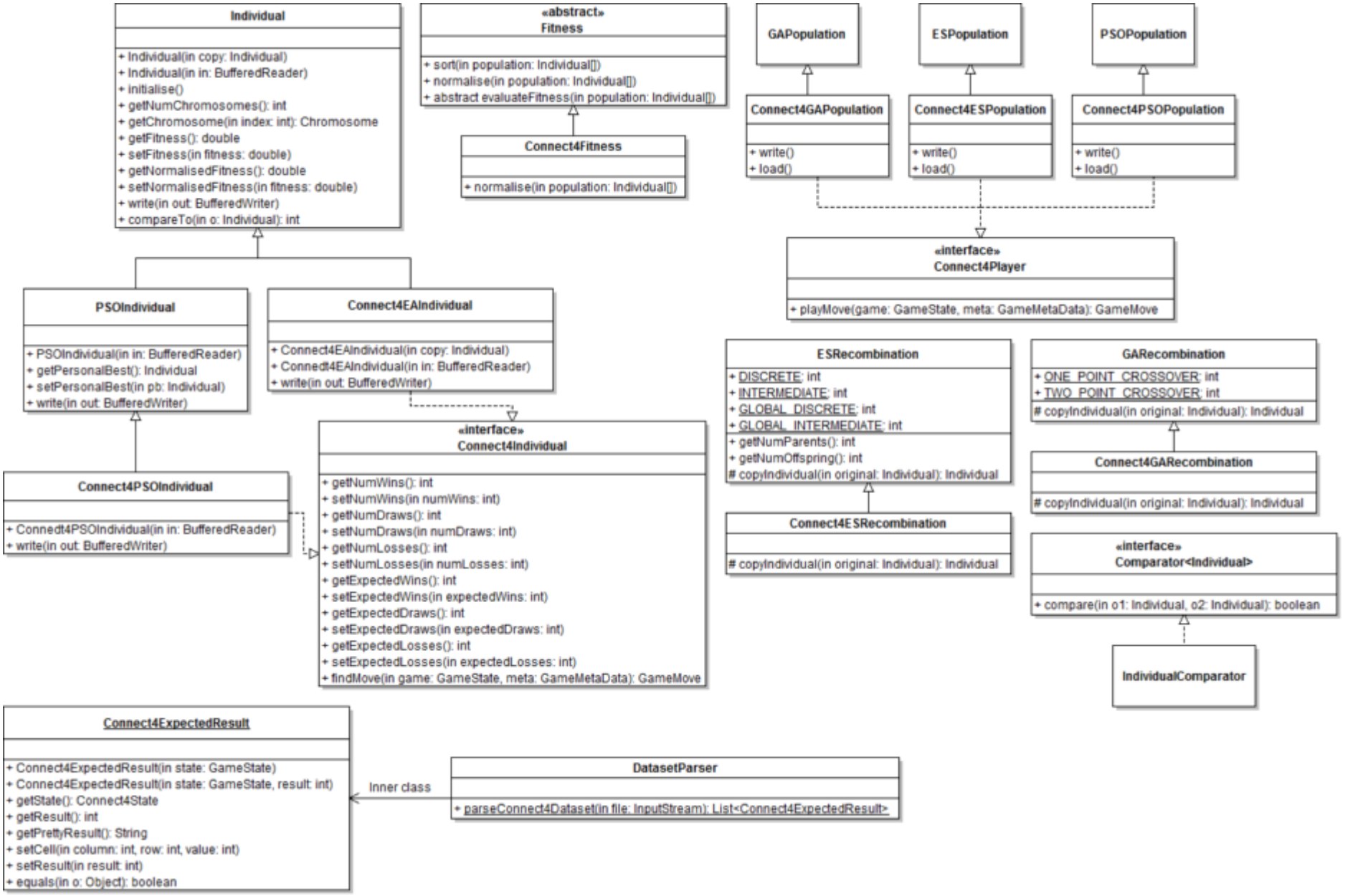


Figure A.2: Class diagram showing the additional Arena classes

Appendix B

Phase One Test Results

B.1 Genetic Algorithm

B.1.1 Binary Tournament Selection With Two-point Crossover

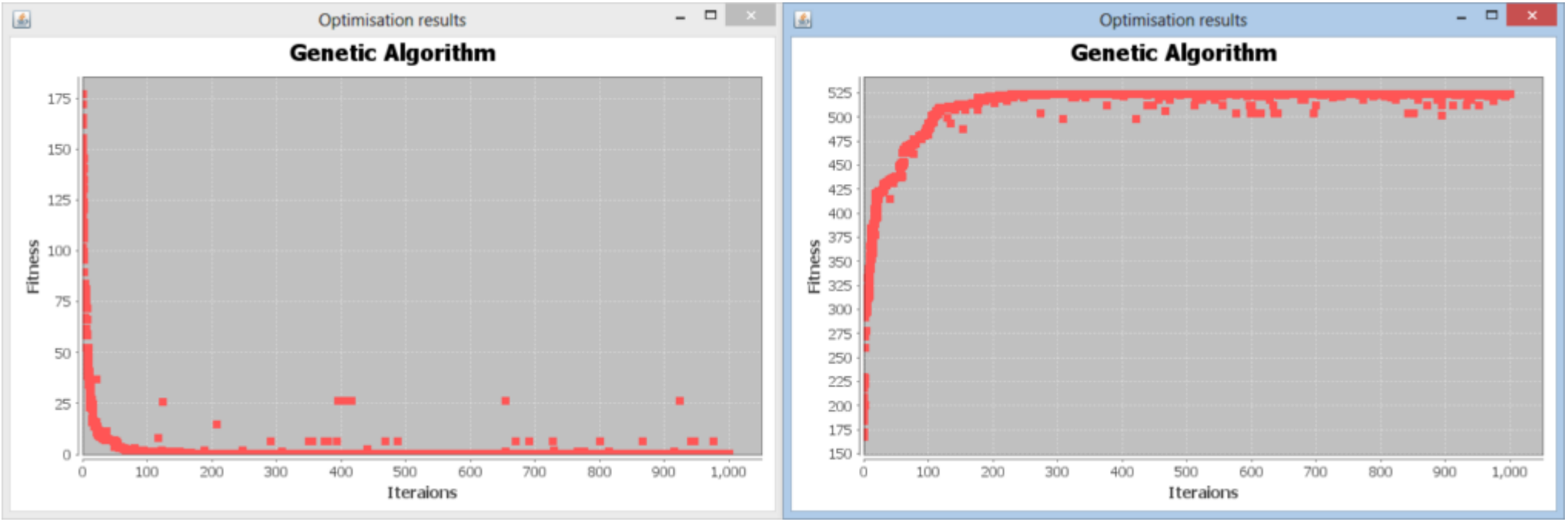


Figure B.1: Optimisation results of a genetic algorithm using binary tournament selection and two-point crossover

B.1.2 Binary Tournament Selection With One-point Crossover

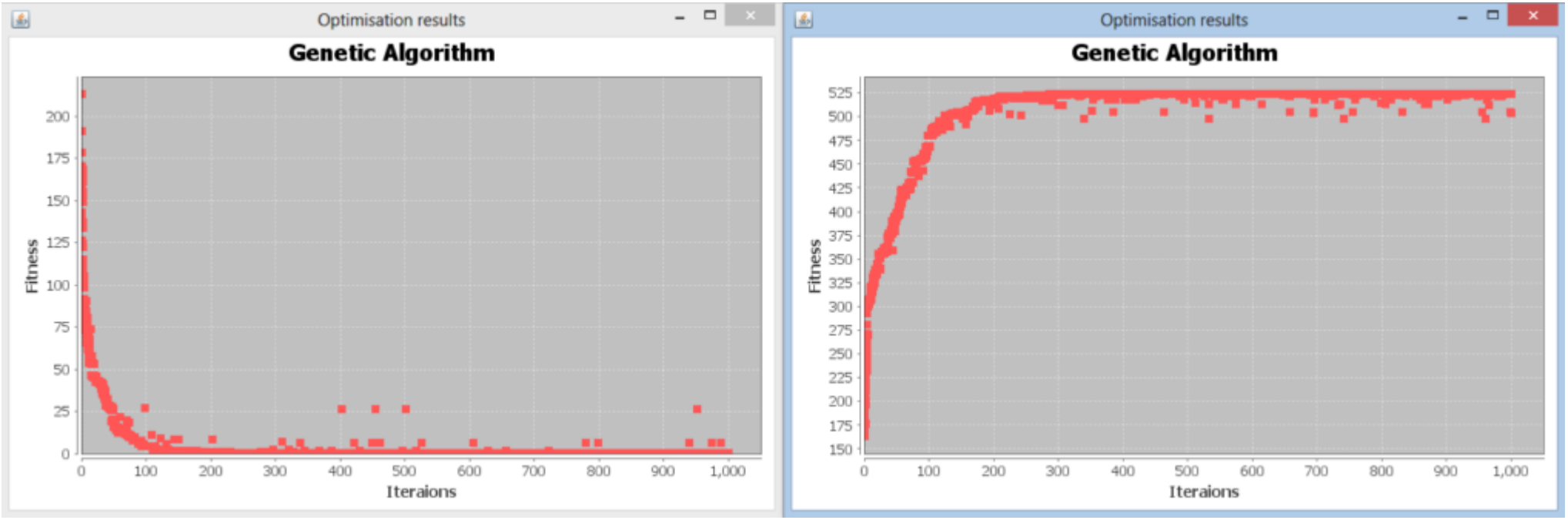


Figure B.2: Optimisation results of a genetic algorithm using binary tournament selection and one-point crossover

B.1.3 Rank Selection With Two-point Crossover

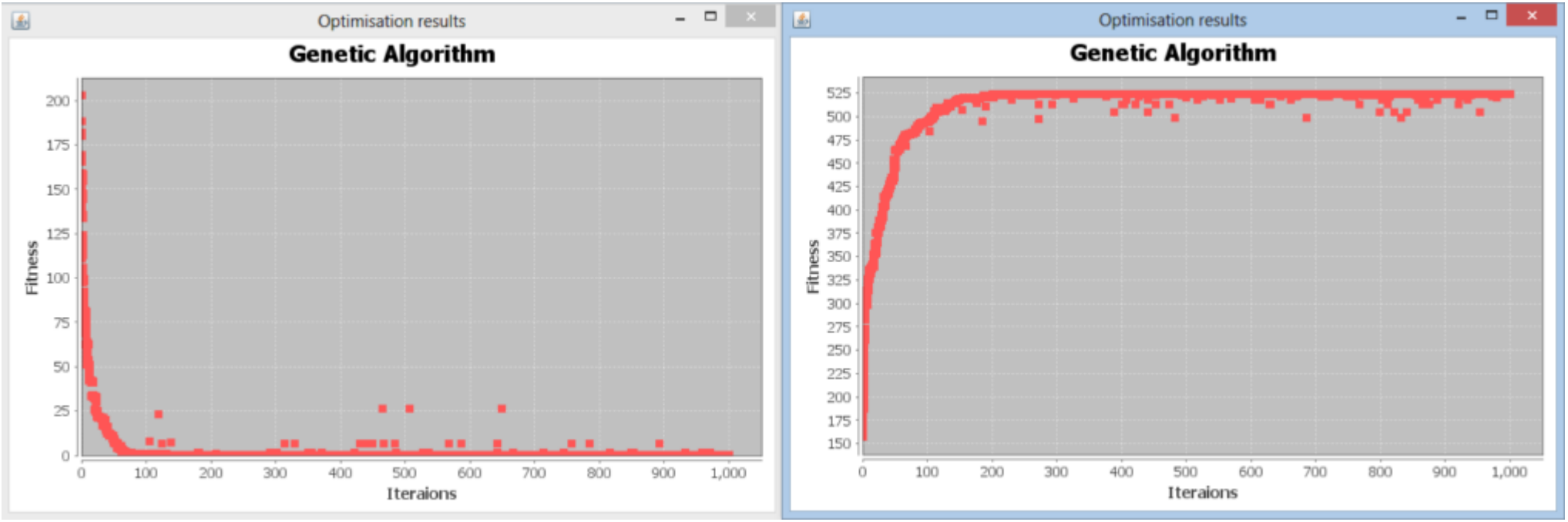


Figure B.3: Optimisation results of a genetic algorithm using rank selection and two-point crossover

B.1.4 Rank Selection With One-point Crossover

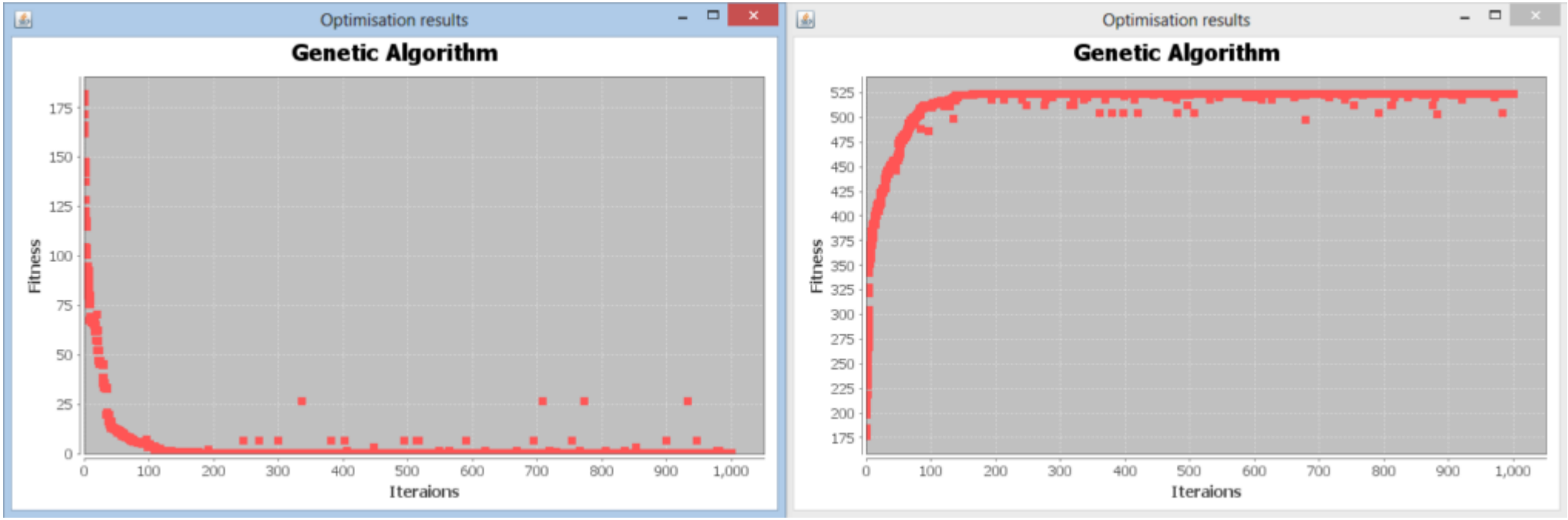


Figure B.4: Optimisation results of a genetic algorithm using rank selection and one-point crossover

B.1.5 Proportional Selection With Two-point Crossover

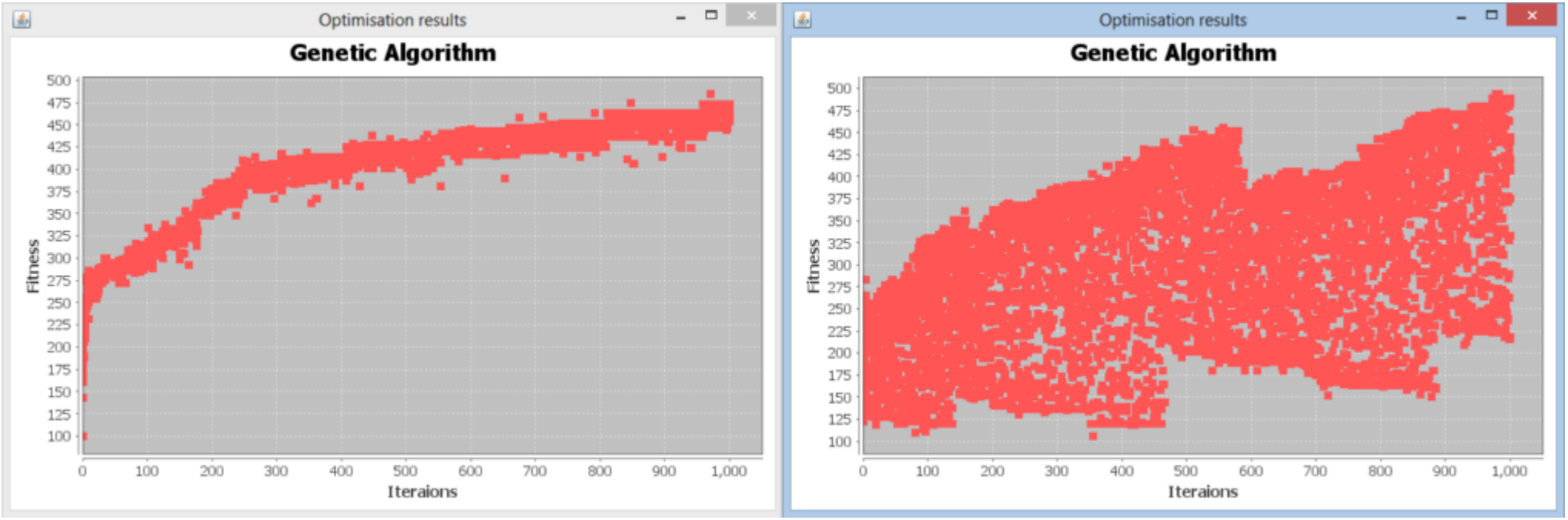


Figure B.5: Optimisation results of a genetic algorithm using proportional selection and two-point crossover

B.1.6 Proportional Selection With One-point Crossover

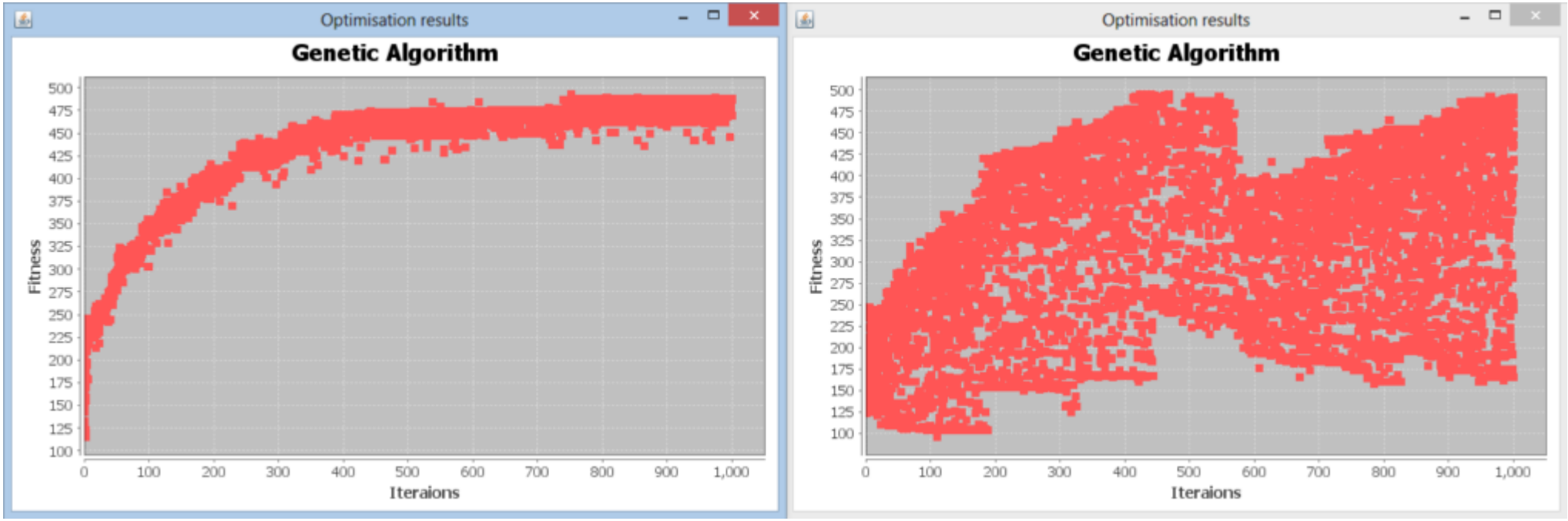


Figure B.6: Optimisation results of a genetic algorithm using proportional selection and one-point crossover

B.1.7 Proportional Selection With Adjusted Numbers of Parents

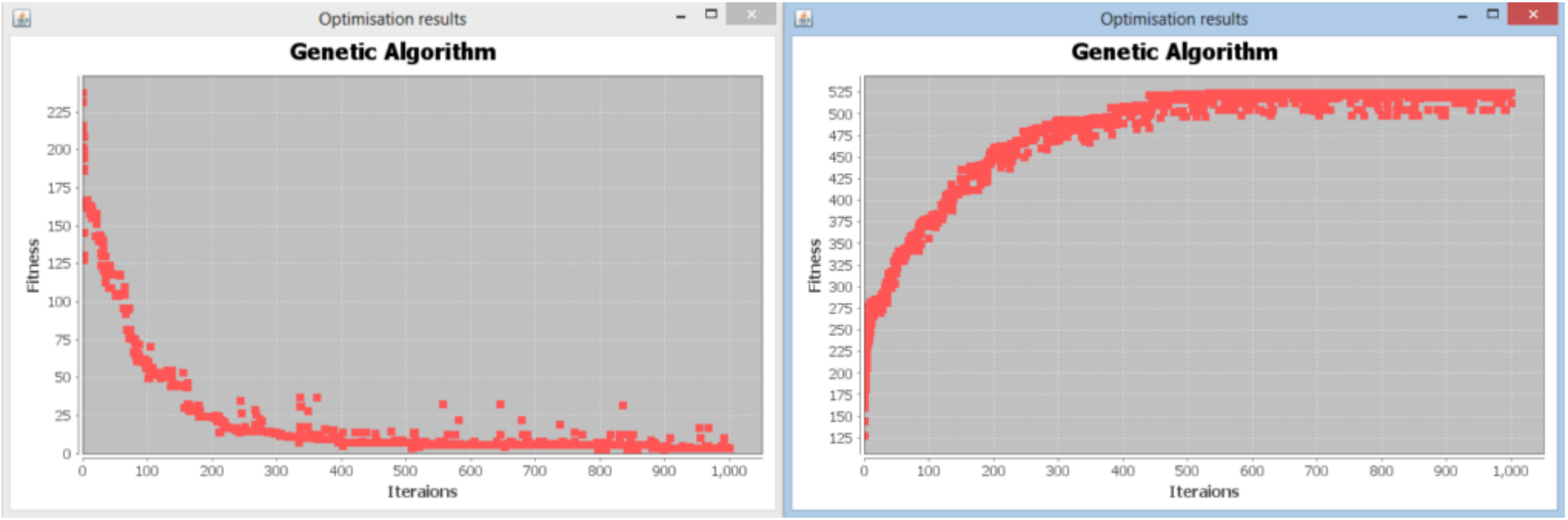


Figure B.7: Optimisation results of a genetic algorithm using proportional selection and two-point crossover. The number of parents has been adjusted to give better results.

B.1.8 Binary Tournament Selection With Two-point Crossover in Shark

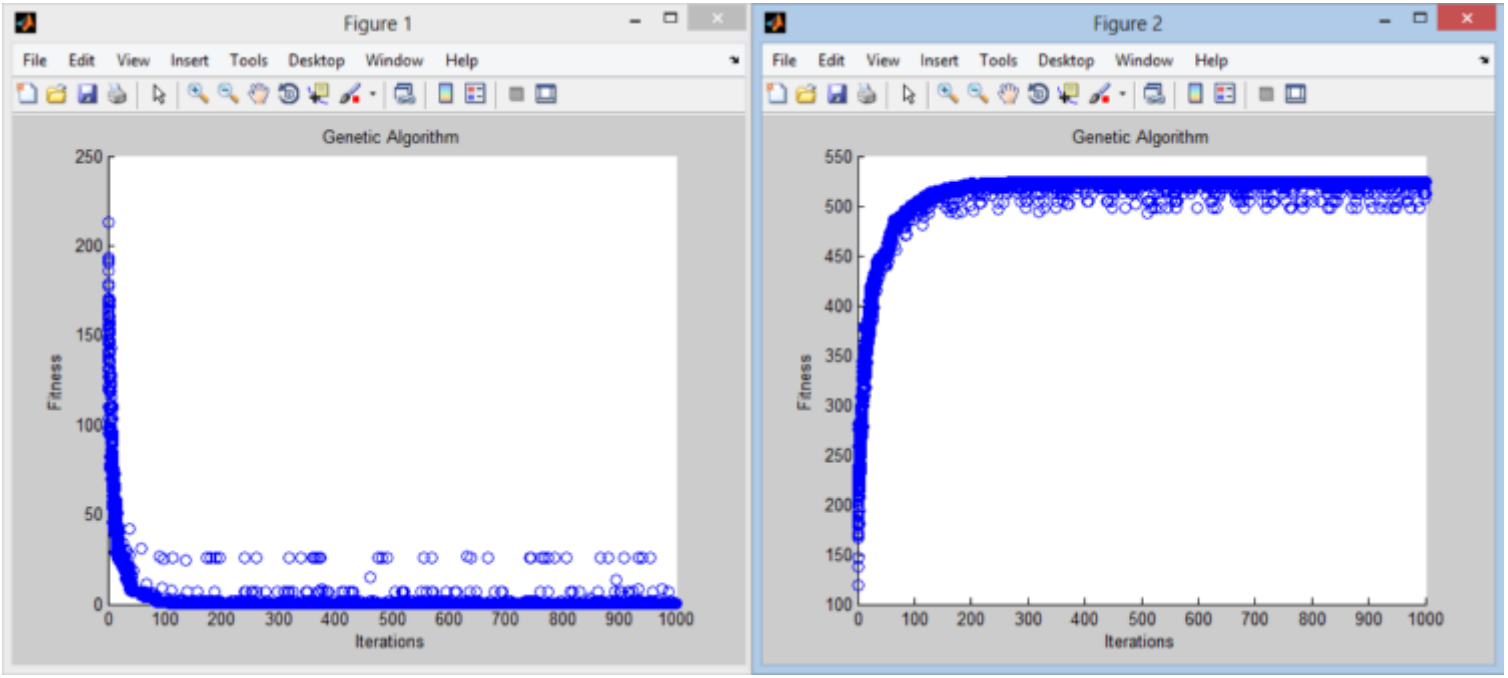


Figure B.8: Optimisation results of a genetic algorithm using binary tournament selection and two-point crossover implemented in Shark

B.2 Evolution Strategies

B.2.1 (μ, λ) Selection With Discrete Recombination

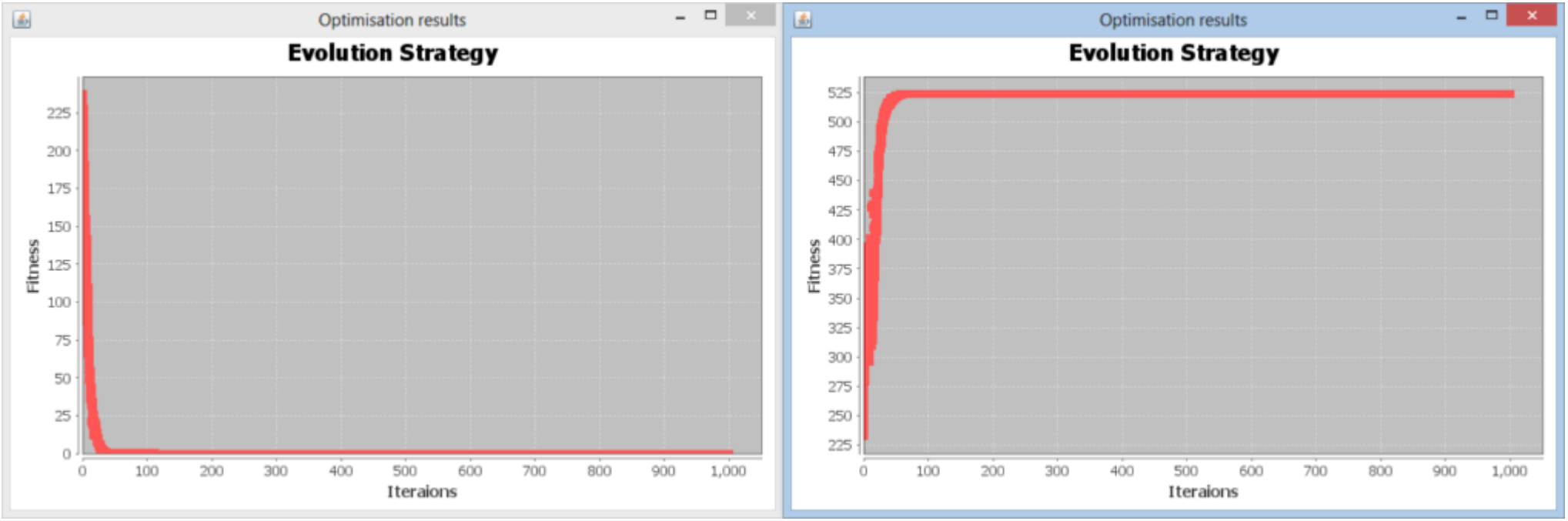


Figure B.9: Optimisation results of evolution strategies using (μ, λ) selection and discrete recombination

B.2.2 (μ, λ) Selection With Intermediate Recombination

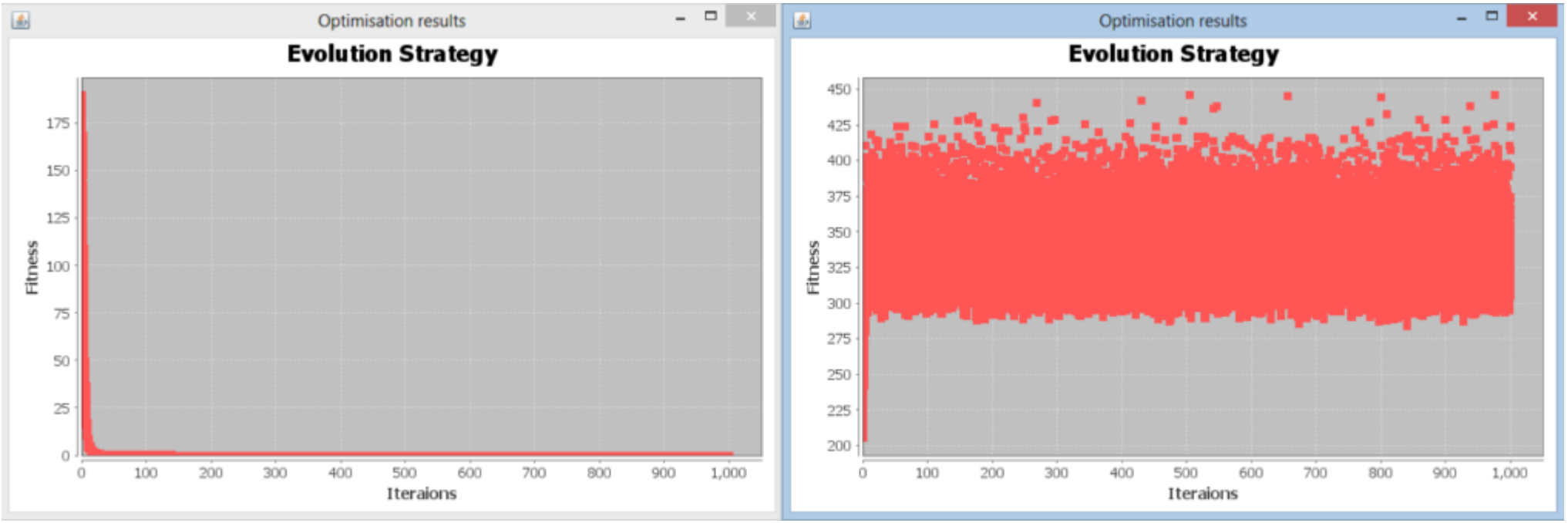


Figure B.10: Optimisation results of evolution strategies using (μ, λ) selection and intermediate recombination

B.2.3 (μ, λ) Selection With Global Discrete Recombination

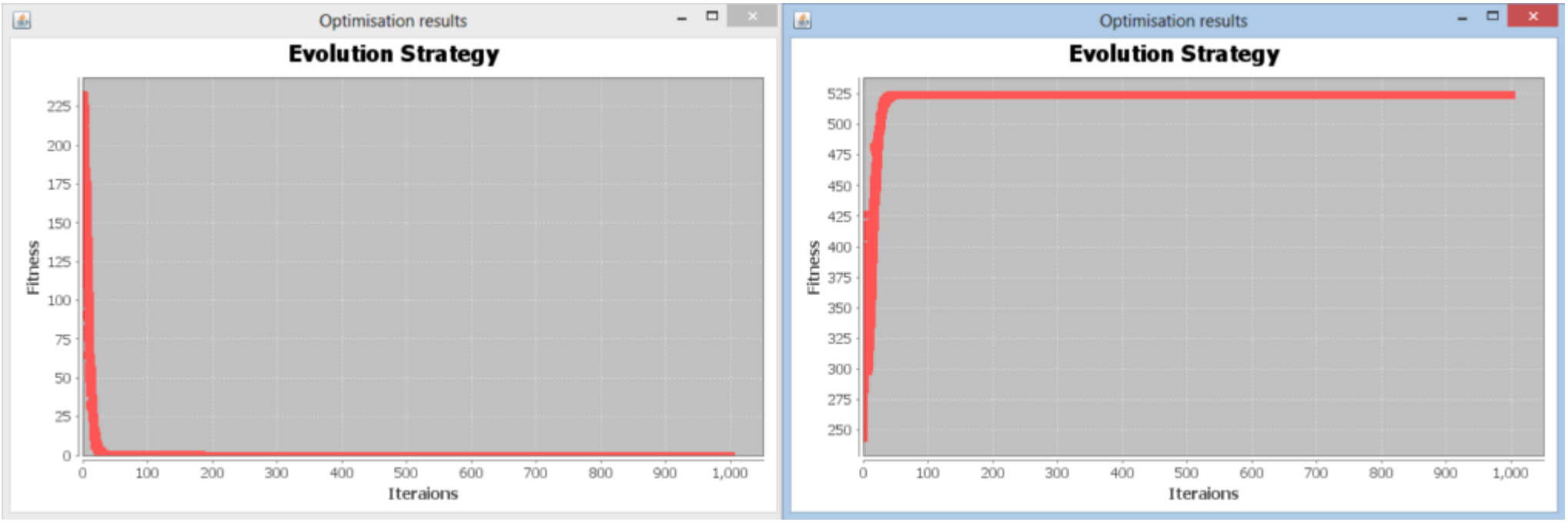


Figure B.11: Optimisation results of evolution strategies using (μ, λ) selection and global discrete recombination

B.2.4 (μ, λ) Selection With Global Intermediate Recombination

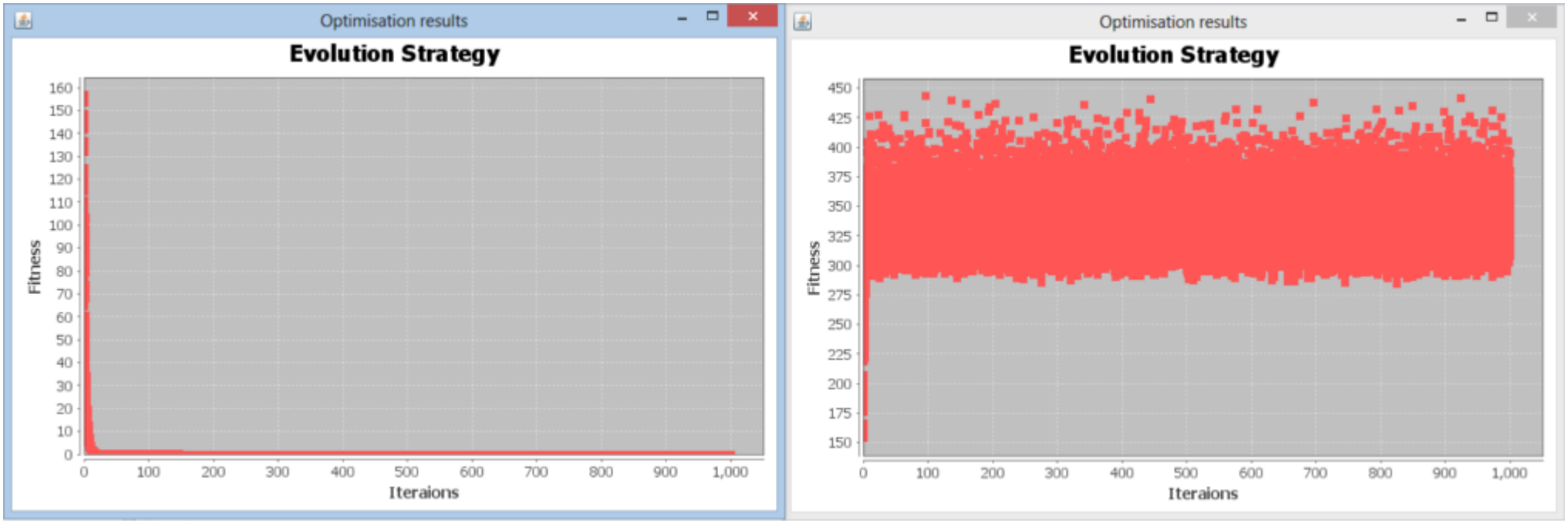


Figure B.12: Optimisation results of evolution strategies using (μ, λ) selection and global intermediate recombination

B.2.5 $(\mu + \lambda)$ Selection With Discrete Recombination

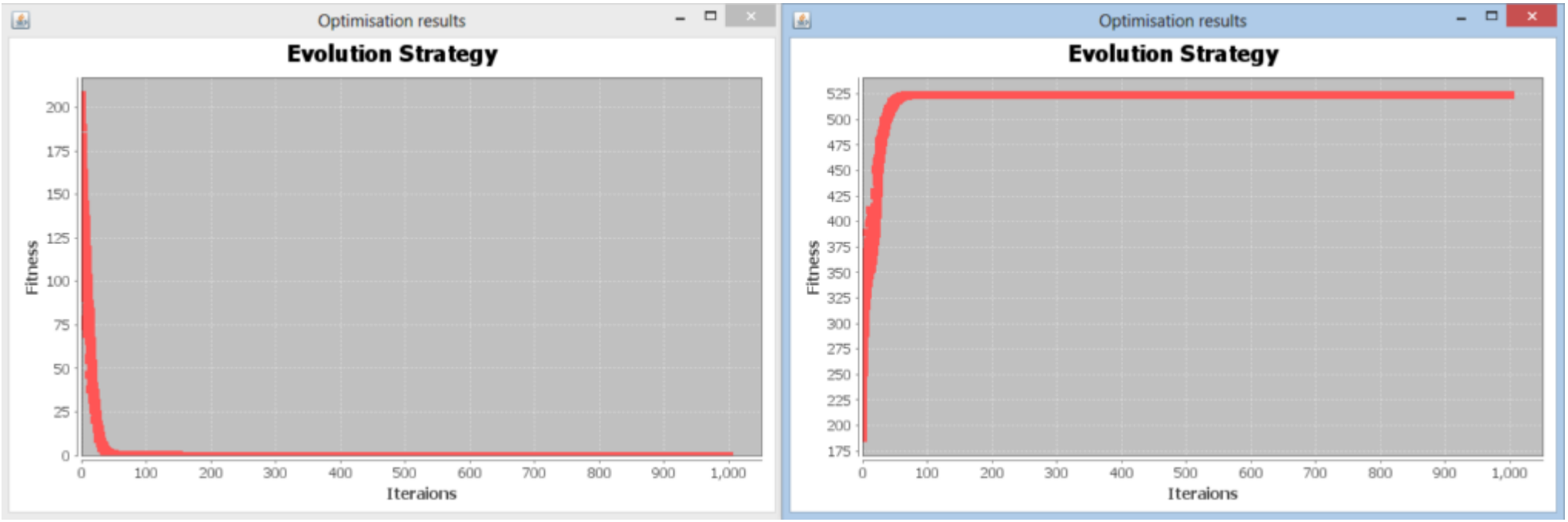


Figure B.13: Optimisation results of evolution strategies using $(\mu + \lambda)$ selection and discrete recombination

B.2.6 $(\mu + \lambda)$ Selection With Intermediate Recombination

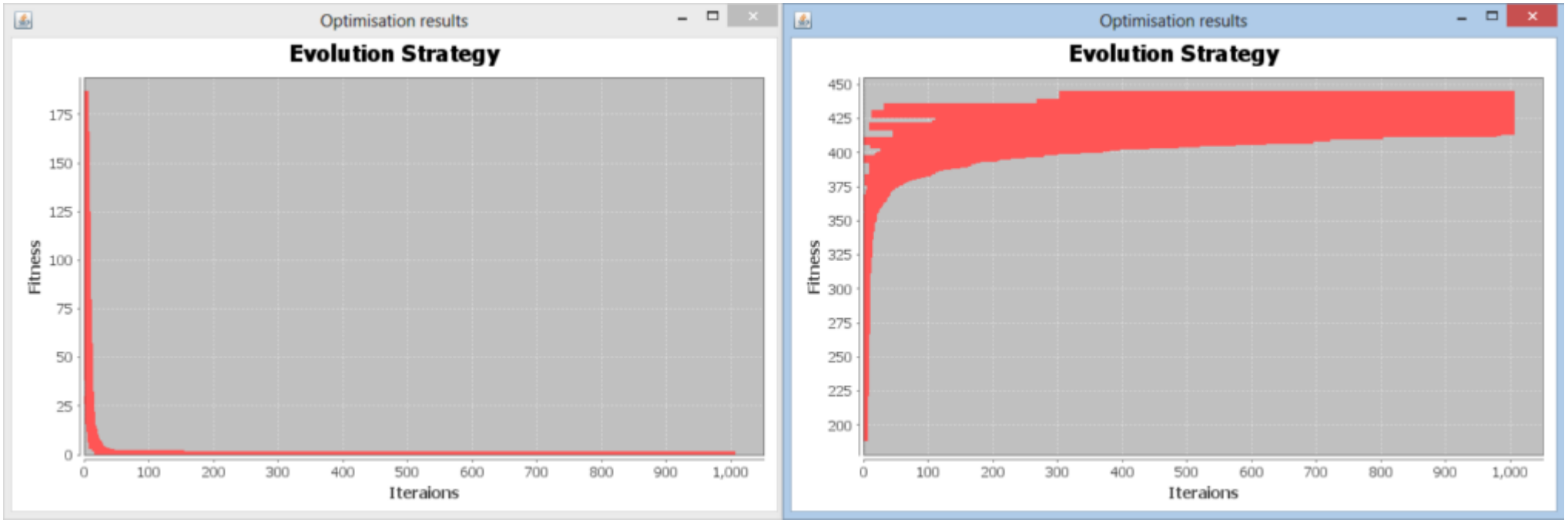


Figure B.14: Optimisation results of evolution strategies using $(\mu + \lambda)$ selection and intermediate recombination

B.2.7 $(\mu + \lambda)$ Selection With Global Discrete Recombination

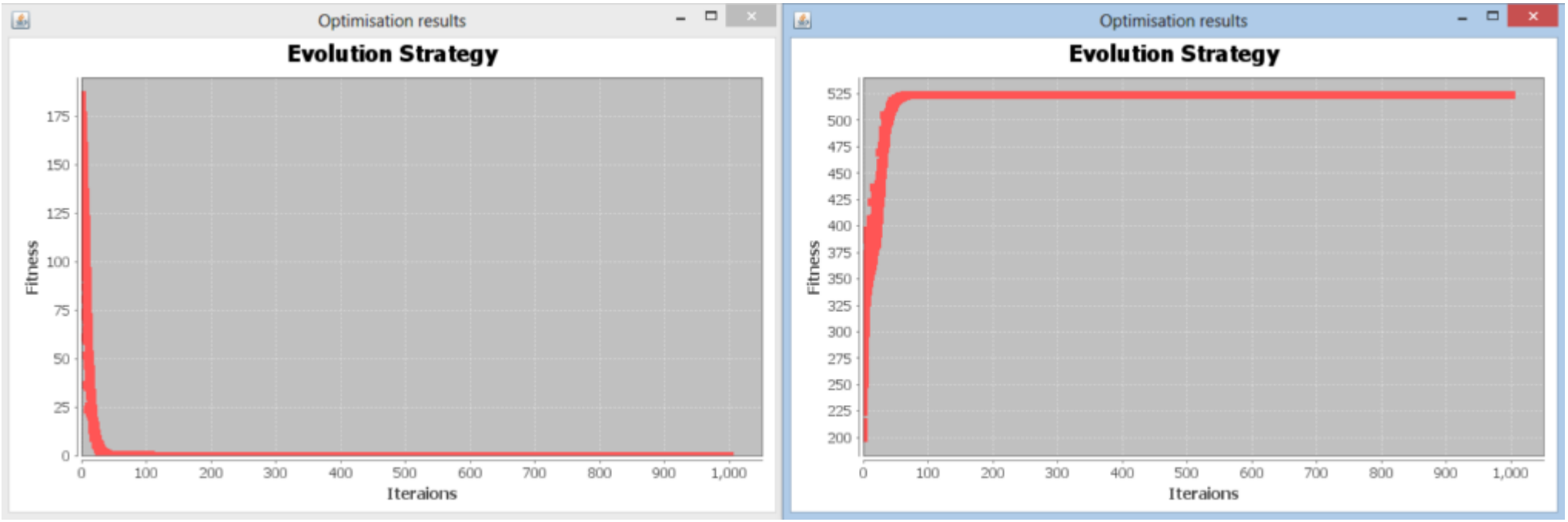


Figure B.15: Optimisation results of evolution strategies using $(\mu + \lambda)$ selection and global discrete recombination

B.2.8 $(\mu + \lambda)$ Selection With Global Intermediate Recombination

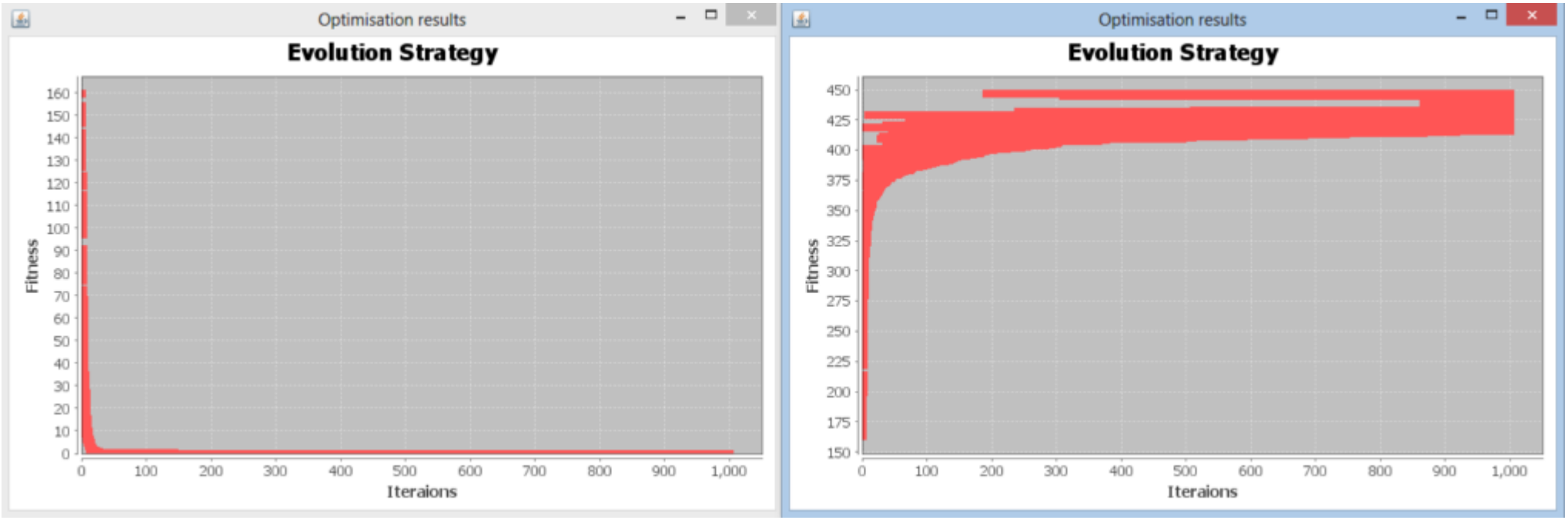


Figure B.16: Optimisation results of evolution strategies using $(\mu + \lambda)$ selection and global intermediate recombination

B.2.9 $(\mu + \lambda)$ Selection With Discrete Recombination in Shark

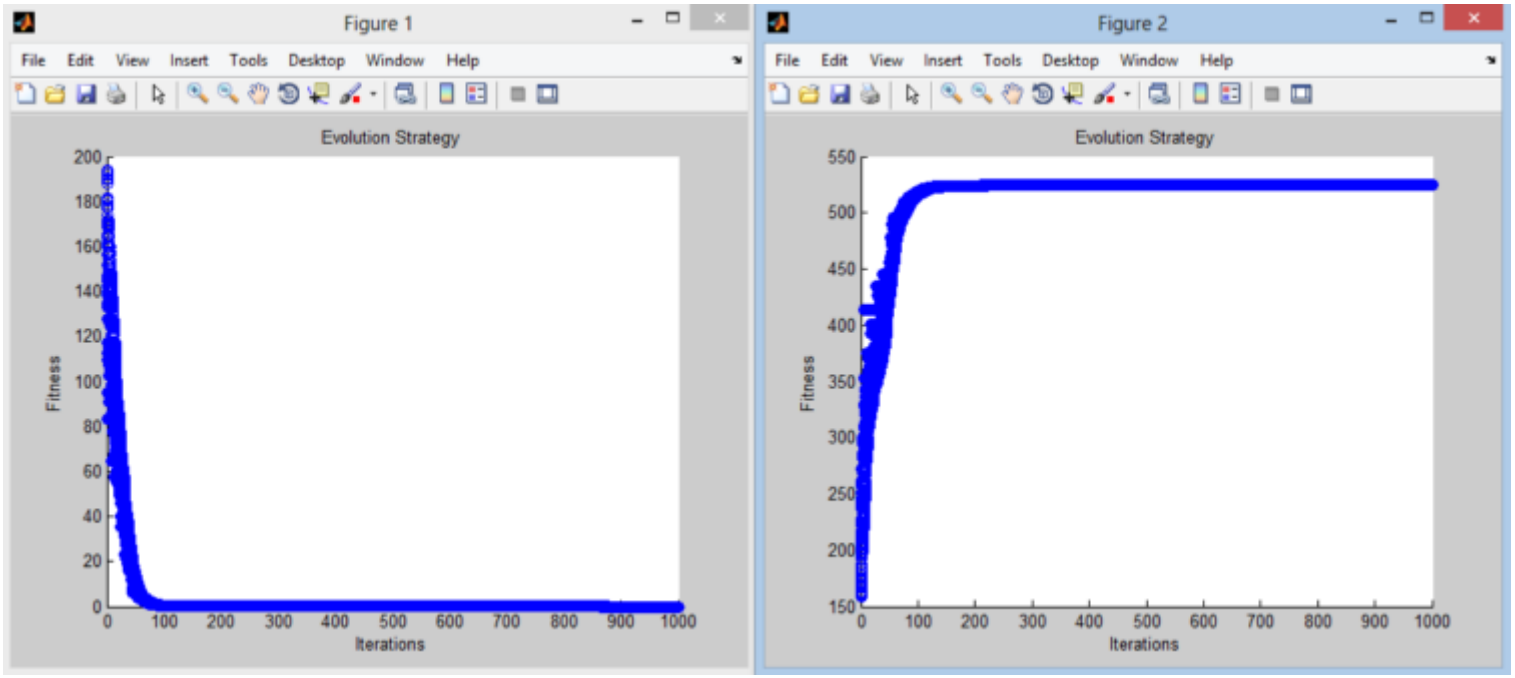


Figure B.17: Optimisation results of evolution strategies using $(\mu + \lambda)$ selection and discrete recombination implemented in Shark

B.3 Particle Swarm Optimisation

B.3.1 Optimisation Results

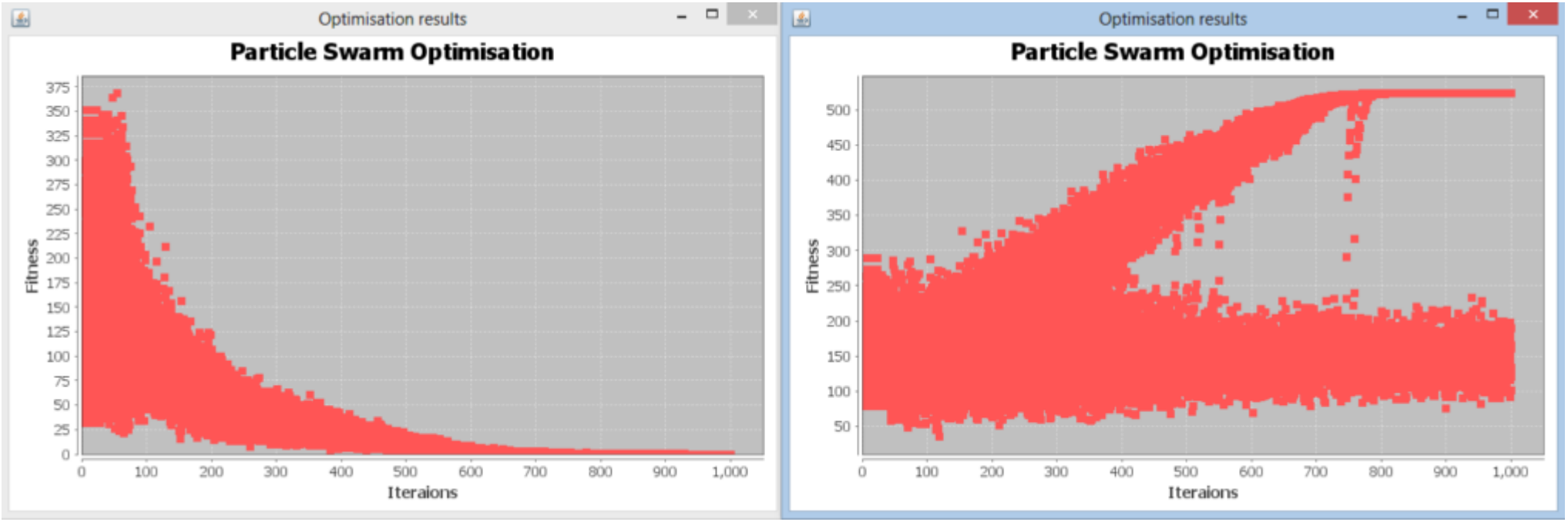


Figure B.18: Optimisation results of particle swarm optimisation

B.3.2 Optimisation Results in Shark

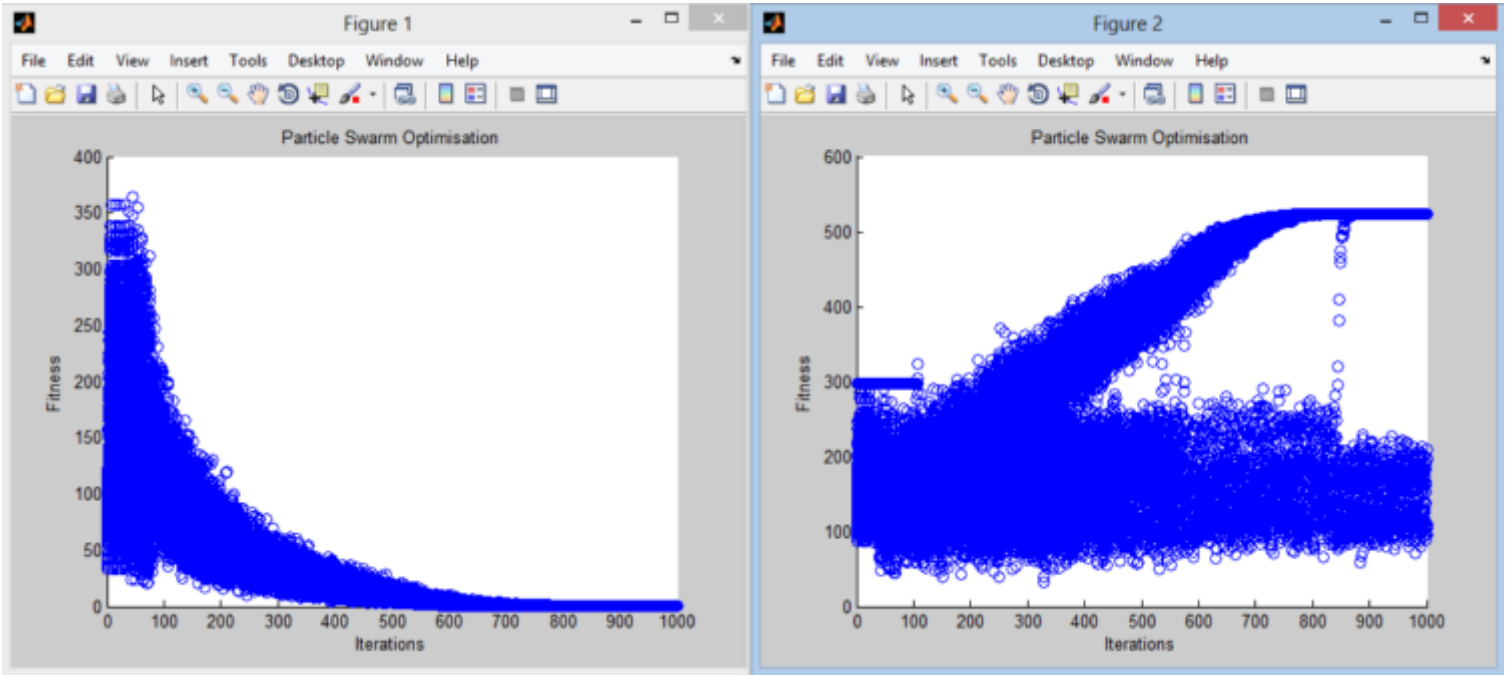


Figure B.19: Optimisation results of particle swarm optimisation implemented in Shark

Bibliography

[1] V. Allis, "A knowledge-based approach of connect-four," Master's thesis, Vrije Universiteit, 1988.

[2] G. B. Danzig. The nature of mathematical programming. [Online]. Available: http://glossary.computing.society.informs.org/index.php?page=nature.html

[3] P. Collet and J. Rennard, *Handbook of Research on Nature Inspired Computing for Economics and Management*, 1st ed. Hershey, 2006, ch. 3, p. 28.

[4] Q. Pan *et al.*, "A self-adaptive global best harmony search algorithm for continuous optimization problems," *Applied Mathematics and Computation*, vol. 216, no. 3, pp. 830–848, Apr. 2010.

[5] M. S. Alam *et al.*, "Diversity guided evolutionary programming: A novel approach for continuous optimization," *Applied Soft Computing*, vol. 12, no. 6, pp. 1693–1707, Jun. 2012.

[6] J. McCarthy. (2007, Nov.) Basic questions. [Online]. Available: http://www-formal.stanford.edu/jmc/whatisai/node1.html

[7] D. Poole *et al.*, *Computational Intelligence A Logical Approach.* Oxford University Press, Jan. 1998, ch. 1, p. 1.

[8] C. A. C. Coello *et al.*, *Evolutionary Algorithms for Solving Multi-Objective Problems*, 2nd ed. Springer, 2007, ch. 1, p. 23.

[9] Ealib documentation. [Online]. Available: http://shark-project.sourceforge.net/EALib/index.html

[10] C. Spieth. (2013) Evolutionary algorithms. [Online]. Available: http://www.ra.cs.uni-tuebingen.de/software/JCell/tutorial/ch03s05.html

[11] A. P. Engelbrecht, *Computational Intelligence An Introduction*, 2nd ed. Wiley, 2007, ch. 9.

[12] A. Marczyk. (2004, Apr.) Genetic algorithms and evolutionary computation. [Online]. Available: http://www.talkorigins.org/faqs/genalg/genalg.html#history

[13] R. L. Haupt and S. E. Haupt, *Practical Genetic Algorithms*, 2nd ed. Wiley, 2004, ch. 1, pp. 22–23.

[14] A. P. Engelbrecht, *Computational Intelligence An Introduction*, 2nd ed. Wiley, 2007, ch. 8.2.

[15] ——, *Computational Intelligence An Introduction*, 2nd ed. Wiley, 2007, ch. 8.5.

[16] B. Zhang and J. Kim, "Comparison of selection methods for evolutionary optimization," *Evolutionary Optimization An International Journal on the Internet*, vol. 2, no. 1, pp. 55–70, 2000.

[17] T. Blickle and L. Thiele, "A comparison of selection schemes used in genetic algorithms," Swiss Federal Institute of Technology, Tech. Rep., 1995.

[18] N. M. Razali and J. Geraghty, "Genetic algorithm performance with different selection strategies in solving tsp," in *Proc. of the World Congress on Engineering*, vol. 2, London, Jul. 2011.

[19] Roulette wheel selection. [Online]. Available: http://www.edc.ncl.ac.uk/highlight/rhjanuary2007g02.php/

[20] Y. Jin, "Com3013: Computational intelligence week 2: Evolutionary algorithms."

[21] H. Sadeghi *et al.*, "A two-point, three-dimensional seismic ray tracing using genetic algorithms," *Physics of the Earth and Planetary Interiors*, vol. 113, no. 1-4, Jun. 1999.

[22] W. Lin *et al.*, "Adapting crossover and mutation rates in genetic algorithms," *Journal of Information Science and Engineering*, vol. 19, 2003.

[23] A. J. Scholand. Genetic algorithm (ga) parameter settings. [Online]. Available: http://eislab.gatech.edu/people/scholand/gapara.htm

[24] M. Mitchell, *An Introduction to Genetic Algorithms*, 1st ed. Massachusetts Institute of Technology, 1998, ch. 5.6, pp. 175–177.

[25] A. P. Engelbrecht, *Computational Intelligence An Introduction*, 2nd ed. Wiley, 2007, ch. 12.

[26] H. Bayer and H. Schwefel, "Evolution strategies a comprehensive introduction," *Natural Computing*, vol. 1, pp. 3–52, 2002.

[27] O. Kramer and H. Schwefel, "On three new approaches to handle constraints within evolution strategies," *Natural Computing*, Nov. 2006.

[28] Z. Michalewicz, "A survey of constraint handling techniques in evolutionary computation methods," in *Proc. of the 4th Annual Conference on Evolutionary Programming*, 1995, pp. 135–155.

[29] C. A. C. Coello, "Constraint-handling techniques used with evolutionary algorithms," in *Proc. of the 2008 GECCO conference companion on Genetic and evolutionary computation*, 2008, pp. 2445–2466.

[30] O. Kramer, "A review of constraint-handling techniques for evolution strategies," *Applied Computational Intelligence and Soft Computing*, Jan. 2010.

[31] A. P. Engelbrecht, *Computational Intelligence An Introduction*, 2nd ed. Wiley, 2007, ch. IV.

[32] ——, *Computational Intelligence An Introduction*, 2nd ed. Wiley, 2007, ch. 15.

[33] M. Reyes-Sierra and C. A. C. Coello, "Multi-objective particle swarm optimizers: A survey of the state-of-the-art," *International Journal of Computational Intelligence Research*, vol. 2, no. 3, 2006.

[34] J. Blondin. Particle swarm optimisation: A tutorial. [Online]. Available: http://cs.armstrong.edu/saad/csci8100/pso_tutorial.pdf

[35] M. Settles. (Nov.) An introduction to particle swarm optimizationl. [Online]. Available: http://www2.cs.uidaho.edu/~tsoule/cs504/particleswarm.pdf

[36] P. Umapathy *et al.*, "Particle swarm optimization with various inertia weight variants for optimal power flow solution," *Discrete Dynamics in Nature and Society*, 2010.

[37] N. Padhye, "Development of efficient particle swarm optimizers and bound handling methods," Master's thesis, Indian Institute of Technology, 2010.

[38] Ecj. [Online]. Available: http://cs.gmu.edu/~eclab/projects/ecj/

[39] Computation intelligence library. [Online]. Available: http://www.cilib.net/

[40] Java genetic algorithm library. [Online]. Available: http://jgal.sourceforge.net/

[41] Java genetic algorithm package. [Online]. Available: http://jgap.sourceforge.net/

[42] Jswarm-pso. [Online]. Available: http://jswarm-pso.sourceforge.net/

[43] Shark machine library. [Online]. Available: http://image.diku.dk/shark/sphinx_pages/build/html/index.html

[44] Shark. [Online]. Available: http://sourceforge.net/projects/shark-project/files/Shark%20Core/

[45] [Online]. Available: http://2.bp.blogspot.com/-92qqLLPAeiw/TnhYfK54fqI/AAAAAAAAAAc/Xz3AaoD00I8/s1600/Picture3.jpg

[46] J. Brownlee, *Clever Algorithms: Nature-Inspired Programming Recipes*, 1st ed. Lulu Enterprises, 2011, ch. 5.1.

[47] Jfree. [Online]. Available: http://www.jfree.org/index.html

[48] H. Pohlheim. (2006, Dec.) Geatbx: Example functions (single and multi-objective functions) 2 parametric optimization. [Online]. Available: http://www.geatbx.com/docu/fcnindex-01.html#TopOfPage

[49] J. Tromp. (1995, Feb.) Connect-4 data set. [Online]. Available: http://archive.ics.uci.edu/ml/datasets/Connect-4

[50] J. Heather. Strategy games competition. [Online]. Available: http://www.computing.surrey.ac.uk/personal/st/J.Heather/personal/strategy/